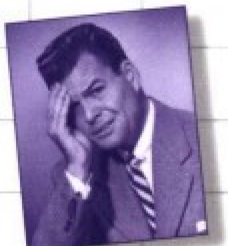
重视大脑的学习指南

Head First
设计模式
（中文版）

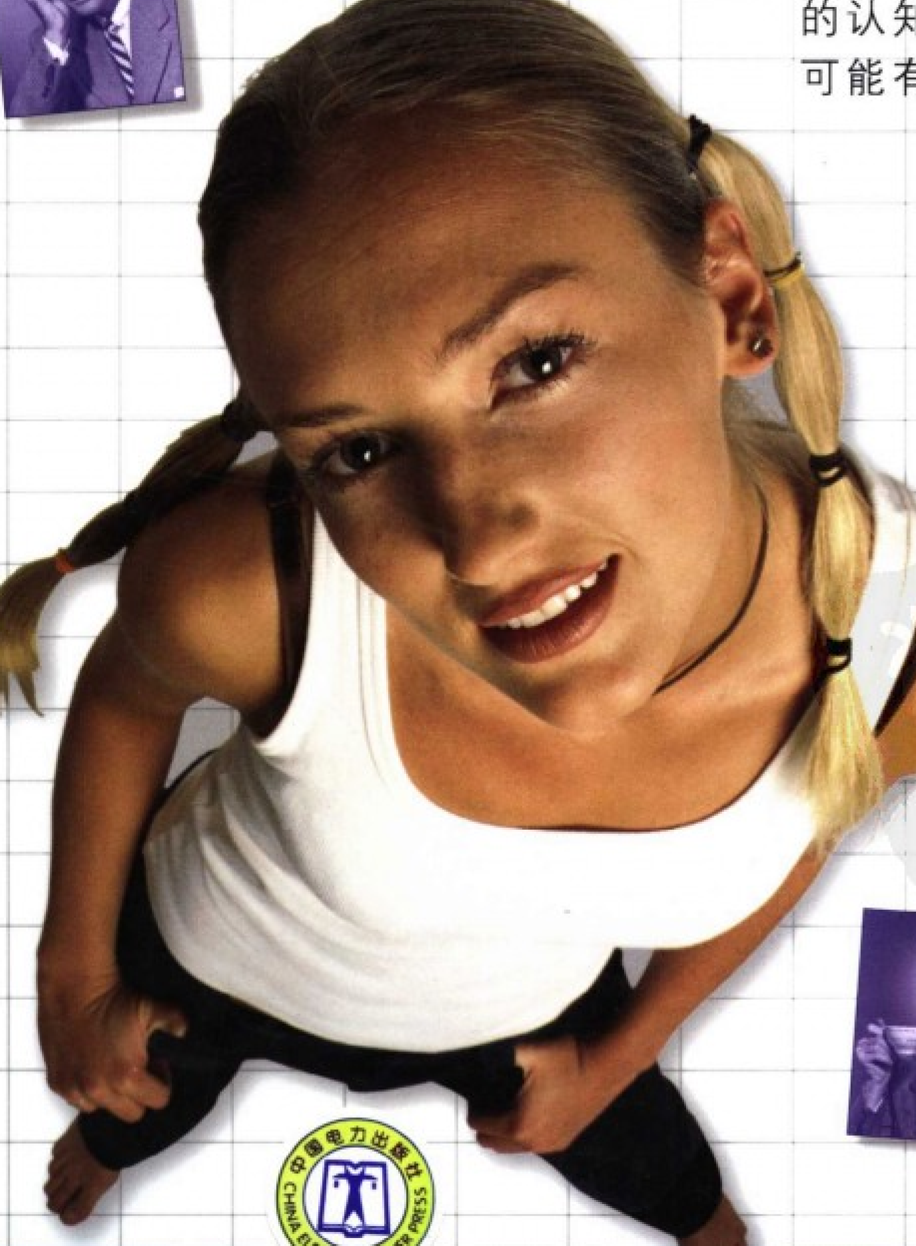


避免一些
尴尬的
耦合错误

学习为何朋友们
对工厂模式
的认知
可能有错

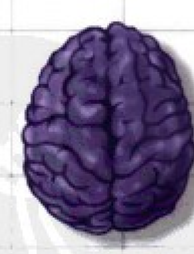






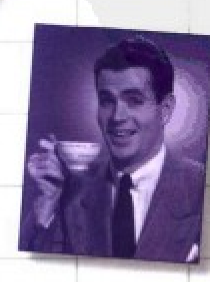
发掘模式大师
的秘密

把事关紧要的模式
直接装入脑海里





探究星巴兹咖啡
如何以装饰者模式
让自己的股价翻倍



瞧瞧Jim
为何拒绝继承后
改善了爱情生活

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates 著
O'Reilly Taiwan公司 译　UMLChina 改编

O'REILLY® 中国电力出版社

Head First设计模式（中文版）

Software Development/Java

你不想重新发明轮子（或者更差的是，漏气的轮子），所以你从设计模式中寻求协助——设计模式是过去人们面对同样的软件设计问题所学来的经验。有了设计模式，你就可以利用他人实践经验的精华，省下的时间可以用在……其他的事情上，一些更有挑战性的事情、更复杂的事情、更有趣的事情。你想要学习：

- 事关紧要的模式
- 何时使用某个模式，为何使用该模式
- 如何在自己的设计中马上采用这些模式
- 何时不该使用模式（如何避免对模式过度狂热）
- 模式是基于哪些面向对象设计原则而设计出来的

更重要的是，你在学习设计模式的过程中不会感到昏昏欲睡。如果你曾经读过任何一本Head First系列书籍，就知道你能够从本书中得到的是：透过丰富的视觉效果让你的大脑充分地工作。本书的编写运用了许多最新的研究，包括神经生物学、认知科学，以及学习理论，这使得这本书能将这些设计模式深深地烙在你的脑海中，不容易被遗忘。你将更擅长于解决软件设计中的问题，并能够和你的团队成员用模式的术语沟通。

Eric Freeman和Elisabeth Freeman是作家、讲师，以及技术顾问。原本在迪士尼公司领导了四年的数字媒体，以及Internet的开发，后来，他们将这些经验应用在他们自己的媒体中，包括本书。Eric具有耶鲁大学的计算机科学博士学位，Elisabeth具有耶鲁大学的计算机科学硕士学位。

Kathy Sierra（javaranch.com的创始者）和Bert Bates是畅销的Head First系列书籍的创立者，也是Sun公司Java开发员认证考试的开发者。

“我昨天收到了这本书后就开始读……我简直欲罢不能。这真是太酷了！不但有趣，涵盖面广，而且切中要点。这本书让我感到印象深刻。”

— *Erich Gamma*,
IBM 杰出工程师、
《设计模式》作者之一

“我感到读这本书的效果等同于读一千磅重的同类书的效果。”

— *Ward Cunningham*,
Wiki 发明者、
Hillside Group 创始人

“本书趋近完美，因为它在提供专业知识的同时，仍然具有相当高的可读性。口吻权威、阅读轻松。”

— *David Gelernter*,
耶鲁大学计算机科学系教授

“这是我阅读过的最有趣且最聪明的软件设计书籍之一。”

— *Aaron LaBerge*,
ESPN.com 技术副主席

本书荣获2005年第十五届Jolt通用类图书震撼大奖。



O'REILLY®
www.oreilly.com

ISBN 978-7-5083-5393-7
9 787508 353937 >
定价：98.00元

O'Reilly Media, Inc.授权中国电力出版社出版

TP311.5/195
2007

Head First设计模式

（中文版）



Eric Freeman,
Elisabeth Freeman,
Kathy Sierra &
Bert Bates 著
O'Reilly Taiwan公司 译
UMLChina 改编

O'REILLY®

Beijing • Cambridge • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Head First设计模式（中文版）/（美）弗里曼（Freeman, E.）等著；O'Reilly Taiwan公司译，UMLChina改编－北京：中国电力出版社，2007

书名原文：Head First Design Pattern

ISBN 978-7-5083-5393-7

I. H… II.①弗… ②O… ③U… III. 软件设计 IV. TP311.5

中国版本图书馆CIP数据核字（2007）第031335号

北京版权局著作权合同登记

图字：10-2005-5994号

©2004 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2007. Authorized translation of the English edition, 2004 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc.出版2004。

简体中文版由中国电力出版社出版，2007。英文原版的翻译得到 O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/	Head First设计模式（中文版）
书　　号/	ISBN 978-7-5083-5393-7
责任编辑/	胡顺增，张旻
封面设计/	Ellie Volckhausen，张健
出版发行/	中国电力出版社（www.infopower.com.cn）
地　　址/	北京三里河路6号（邮政编码100044）
印　　刷/	北京盛通印刷股份有限公司
开　　本/	787毫米×980毫米 16开本 42.5印张 928千字
版　　次/	2007年9月第1版 2007年9月第1次印刷
印　　数/	0001－5000册
定　　价/	98.00元（册）

敬告读者

本书封面贴有防伪标签，加热后中心图案消失。

本书如有印装质量问题，我社发行部负责退换。

版权所有　翻印必究

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc.授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc.是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。

译者序

设计模式（Design Pattern）很重要，不需要我多说。你瞧，程序员几乎人手一本四人组（Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides）所著的《设计模式》。打个比喻：信耶稣的人都要读圣经，而信OO的人都要读四人组的《设计模式》，这就是OO的圣经。更有趣的是，有人还不只买这本书的原版书、连它的光盘版、和中译本也一并买了收藏，可见这是一本多么受到重视的书。我打探过这本书的销售量，它畅销的程度令人咋舌。

许多人反映，四人组的《设计模式》不容易阅读。对于不容易阅读的书，会有已经悟道的人写出白话版或注释版，以飨后进。所以圣经和佛经都有注释版，用更白的方式阐述其中的道理，而我认为《Head First 设计模式》也是因应这样的需求而产生，它可以被视为是白话版、搞笑版、漫画版的《设计模式》。《Head First 设计模式》比起《设计模式》好读得多了，内容也相当有趣。相信我，要写出这样的一本书绝对比写一本正儿八经的书难上许多，可见作者煞费苦心。作者的用心换来空前的成功。《Head First 设计模式》得到相当正面的读者响应，连《设计模式》原创者Erich Gamma也慨然为《Head First 设计模式》写一段推荐文来"作保证"。《Head First 设计模式》还得到2005年的Jolt Award大奖，风光至极。

本书大纲

本书共有14章，每章都介绍了几个设计模式，完整地涵盖了四人组版本全部23个设计模式。前言先介绍这本书的用法；第1章到第11章陆续介绍的设计模式为Strategy、Observer、Decorator、Abstract Factory、Factory Method、Singleton、Command、Adapter、Facade、Template Method、Iterator、Composite、State、Proxy。最后三章比较特别。第12章介绍如何将两个以上的设计模式结合起来成为新的设计模式（例如著名的MVC模式），作者称其为复合设计模式（这是作者自创的名称，并非四人组的标准名词），第13章介绍如何进一步学习设计模式，如何发觉新的设计模式等主题，至于第14章则很快地浏览尚未介绍的设计模式，包括Bridge、Builder、Chain of Responsibility、Flyweight、Interpreter、Mediator、Memento、Prototype、Visitor。

第1章还介绍了四个OO基本概念（抽象、封装、继承、多态），而第1章到第9章也陆续介绍了九个OO原则（Principle）。千万不要轻视这些OO原则，因为每个设计模式背后都包含了几个OO原则的概念。很多时候，在设计时有两难的情况，这时候我们必须回归到OO原则，以方便判断取舍。可以这么说：OO原则是我们的目标，而设计模式是我们的做法。

本书特色

强大的写作阵容。本书作者Eric Freeman和Elisabeth Freeman是作家、讲师和技术顾问。Eric拥有耶鲁大学的计算机科学博士学位，Elisabath拥有耶鲁大学的计算机科学硕士学位。Kathy Sierra（javaranch.com的创始人）和Bert Bates是畅销的Head First系列书籍的创立者，也是Sun公司Java开发员认证考试的开发者。

本书的产品设计应用神经生物学、认知科学，以及学习理论，这使得这本书能够将这些知识深深地印在你的脑海里，不容易被遗忘。本书的编写方式采用引导式教学，不直接告诉你该怎么做，而是利用故事当作引子，带领读者思考并想办法解决问题。解决问题的过程中又会产生一些新的问题，再继续思考、继续解决问题，这样可以加深体会。作者以大量的生活化故事当背景，例如第1章是鸭子，第2章是气象站，第 3 章是咖啡店，书中搭配大量的插图（几乎每一页都有图），所以阅读起来生动有趣，不会感觉到昏昏欲睡。作者还利用歪歪斜斜的手写字体，增加“现场感”。精心设计许多爆笑的对白，让学习过程不会太枯燥。还有模式告白节目，将设计模式拟人化成节目来宾，畅谈其内在的一切。

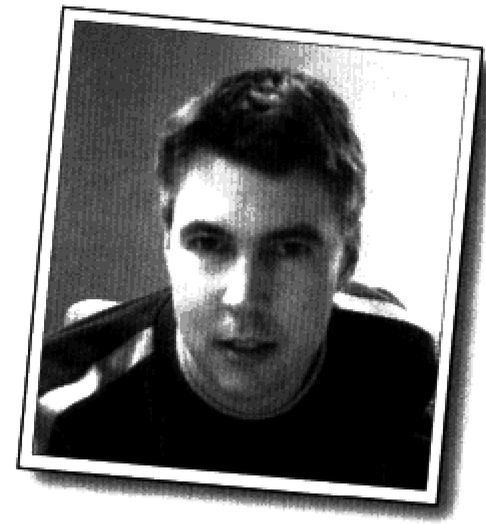
本书大量采用UML的Class Diagram（Static Structure Diagram）。书中的例子程序虽然都是用Java编写，但是本书所介绍的内容对于任何OO语言的用户都适用，包括C++和C#。每一章都有数目不等的测验题。每章最后有一页要点整理，这也是精华所在，我都是利用这一页做复习。

我认为，这本书的作者全都是“变态”！唔，我是说，好的那种“变态”。毕竟要把这么枯燥的主题写得这么有趣而学习效果又好，不是“变态”的作者还真是做不到呢！

《Head First设计模式》的作者/开发者

Elisabeth Freeman





Eric Freeman

Elisabeth是作者、软件开发人员及数字艺术家。她很早就开始进行Internet相关的研究，也是Ada Project的共同发起人（Ada Project是一个针对在计算机界工作的女性而设计的网站，曾获得大奖，现在已经并入ACM）。最近她带领迪士尼的数字媒体研发力量与他人共同发明了一个名为Motion的内容系统，此系统每天传送巨量的数字内容给迪士尼、ESPN及Movies.com的用户。

Elisabeth本质上是一个计算机科学家，拥有耶鲁大学和印第安那大学的计算机科学硕士学位。她的工作领域很广，包括视觉语言、RSS内容整合与Internet系统。她也很积极提倡女性从事计算机工作。今天，你可以发现她在她的Mac上使用Java或Cocoa，但是其实，她最希望的是全世界都使用Scheme。从小在苏格兰长大，Elisabeth喜欢在大自然踏青及户外活动。一旦她在户外，相机总是不离手。她热爱骑单车，是个素食主义者，也很喜欢动物。

她的电子邮件信箱是beth@wickedlysmart.com，你可以发电子邮件给她。

Eric是一个计算机科学家，热衷于软件架构和媒体。

他刚刚花了四年的时间在一个梦寐以求的工作上：在迪士尼指导Internet宽带与无线应用。现在，他回到写作的岗位上，用Java和Mac创造很酷的软件。

在90年代，Eric和David Gelernter一起花了大量的时间，寻找Desktop metaphor的替代品。（他们“仍然”在问：我干嘛不得不给计算机文件取个名字）。也因为这样的研究，Eric在1997年获得耶鲁大学的博士学位。他也与他人一同创立了Mirror Worlds Technologies公司（已经被收购），将他的论文内容商业化，创建了一套软件Lifestreams。

以前，Eric为网络和超级计算机写软件，你可能通过《JavaSpaces Principles Patterns and Practice》这本书得知他的名号。他曾在Thinking Machine CM-5上实现了元组空间系统（tuple-space system），也在80年代末期为NASA创建了第一个Internet信息系统，他为此深感自豪。

Eric目前住在圣达菲附近的沙漠中，当他不写书或代码时，他总是花更多时间摆弄他的家庭影院，而不是观看影片，他利用空档时间试着修复80年代的经典视频游戏Dragon Lair。他也不介意在晚上兼差当个电音DJ。

给他的E-mail可以写到eric@wickedlysmart.com，你也可以去参观他的Blog，网址在http://www.ericfreeman.com。

Head First系列的创立者（以及本书共同策划者）





Kathy自从开始设计游戏以来（她为Virgin、MGM、Amblin等都编写过游戏），一直对学习理论很感兴趣。Head First系列的大多数格式都出自她的手，具体来说，都是她在为UCLA Extension（加利福尼亚大学洛杉矶分校）的“Entertain ment Studies”研究项目教授“New Media Authoring”（新媒体创作）课程时完成的。最近，她成为Sun公司的一名高级培训人员，负责教Sun的Java讲师如何讲授最新的Java技术，并参与开发了多个Sun的认证考试，其中就包括SCBCD考试。与Bert Bates一道，她积极地使用Head First概念来教成千上万的开发人员。她还是世界上最大的Java群体网站javeranch.com的创始人之一，这家网站赢得了2003年和2004年《软件开发》杂志生产力大奖。有时你还会看到她在Java Jam Geek Cruise（geekcruises.com）给学生上Java认证课程。

她最近从加州搬到了科罗拉多，在这里，她得学习一些新的词汇，包括“刨冰机”、“羊绒大衣”（译注），但是在这里的字典里找不到闪电两个字。

喜欢的事：跑步、滑雪、滑板、和她养的冰岛马玩、以及怪力乱神的玩意儿。不喜欢：Entropy（混乱）。

你可以在javaranch.com找到她，偶而她也会出现在java.net的blog中。写给她的信可以寄到kathy@wickedlysmart.com。

译注：　加州会打雷，科罗拉多州会下雪。

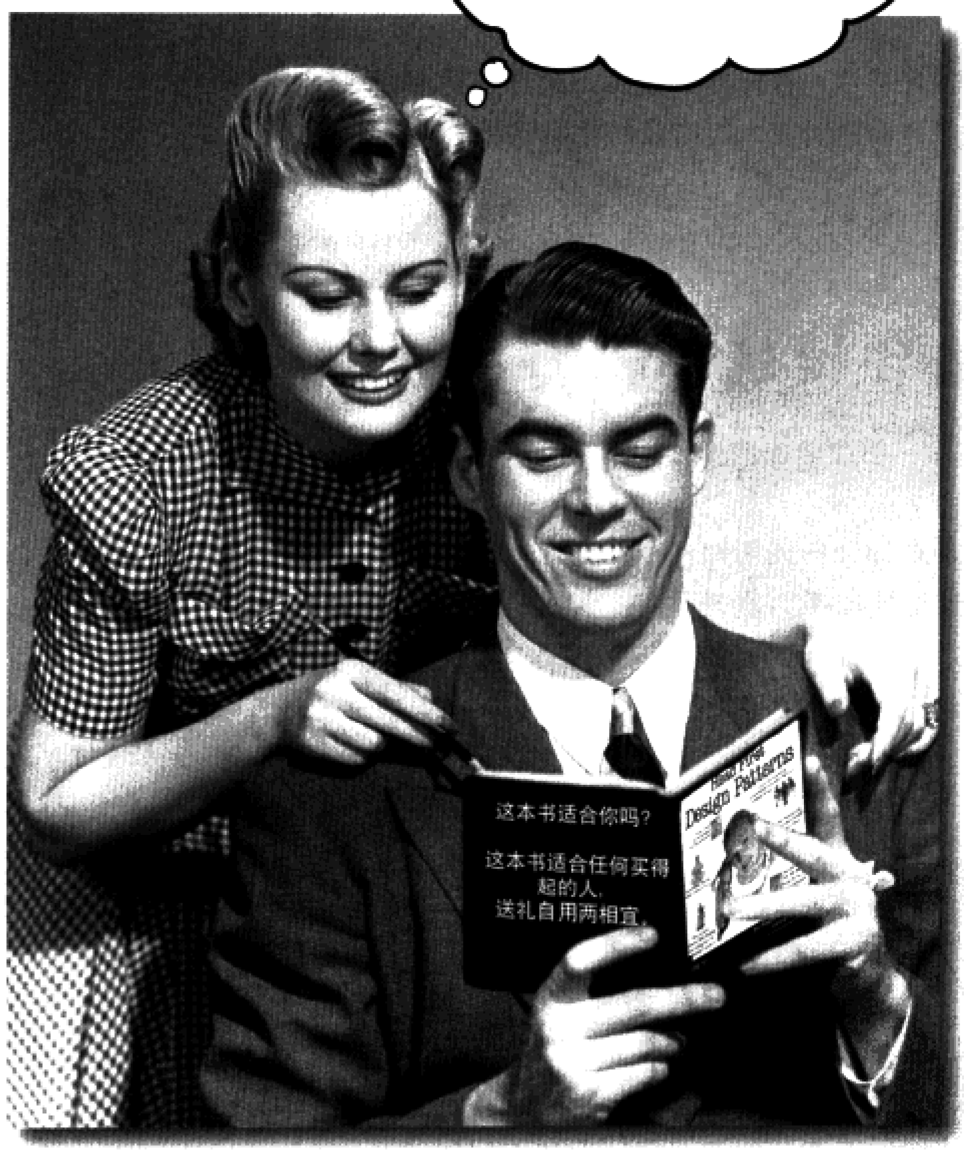
Bert很早就是一位软件开发者和建构师，不过由于在人工智能领域有近十年的经历，使得他对学习理论和基于技术的培训发生了兴趣。从那以后，他一直在教客户学习编程。最近，他成为Sun的Java认证考试开发小组的一员。

在他软件生涯的最初十年，他全世界游历，向Radio New Zealand、Weather Channel和Arts& Entertaininent Network(A&E)这样一些客户提供帮助。他最得意的项目是为Union Pacific Railroad构建了一个全轨系统仿真应用。

长久以来，Bert一直是无可救药的围棋玩家，玩围棋的时间已经长得超乎想象。他的吉他弹得不错，现在更意图染指Banjo（五弦琴或称斑鸠琴）。

你可以在Javaranch.com找到他，或者在IGS go Server上找到他。你也可以通过terrapin@wickedlysmart.com给他写信。

引子



有一个问题真是听得我们耳朵都磨出茧了，这就是：“你们为什么要把这样一些东西放在一本设计模式书里呢？”这一节正是要回答这个问题。

谁适合读这本书？

如果对下面的所有问题你都能肯定地回答“是”：

如果你会C#可能也可以。

1. 你懂Java吗？（不过不要求精通。）

2. 你想学习、了解、记得并应用设计模式，以及其所基于的OO设计原则吗？

3. 你是不是更喜欢一种轻松的氛围，就像在餐桌上交谈一样，而不愿意被动地听技术报告似的枯燥乏味的说教？

那么，本书正是你需要的。

谁暂时还不适合读这本书？

如果对下面任何一个问题你能回答“是”：

1. 你是不是对Java一无所知？
(你不需要是高手，甚至你只会C#但不会Java也没关系，因为两者的相似度是80%。如果你只有C++背景，其实也应该没关系。)

2. 你是不是一个很棒的OO设计者/开发人员，正在找一本参考书？

3. 你是不是一个架构师，想找企业设计模式？

4. 你是不是对新鲜事物都畏手畏脚？你是不是宁愿接受牙根管治疗，也不愿意接受苏格兰花格裙？你是不是觉得，如果把Java组件都拟人化了，这样的一本书肯定不是一本正儿八经的技术书？

那么，太遗憾了，本书不适合你。



[营销备注：本书适合所有有信用卡的人。]

我们知道你在想什么

“这算一本正儿八经的编程书吗？”
“这一堆图是干什么的？”
“我真的能这样学吗？”

我们也知道你的大脑在想什么。



你的大脑总是渴求一些新奇的东西，它一直在搜寻、审视、期待着不寻常的事情发生。大脑的构造就是如此。正是这一点才让我们不至于固步自封，能跟着时代前进。

如今，一般是不太可能被老虎吃掉的。然而，你的大脑还是一直在注意着周围是否有潜伏的老虎。只不过你自己没有意识到而已。但是我们每天都会遇到许多按步就班的事情，这些事情很普通，对于这样一些例行的事情或者平常的东西，你的大脑又是怎么处理的呢？它的做法很简单，就是不让这些平常的东西妨碍大脑真正的工作。那么什么是大脑真正的工作呢？这就是记住那些确实重要的事情。它不会费心地去记乏味的东西；就好像大脑里有一个筛子，这个筛子会筛掉“显然不重要”的东西，如果遇到的事情枯燥乏味，这些东西就无法通过这个筛子。

那么你的大脑怎么知道到底哪些东西重要呢？打个比方，假如你某一天外出旅行，突然一只大老虎跳到你面前，此时此刻，你的大脑里会发生什么呢？

看到这只大老虎，你的神经元会“点火”，情绪爆发，释放出一些化学物质。

好了，这样你的大脑就会知道……

这肯定很重要！可不能忘记了！



不过，假如你正待在家里或者坐在图书馆里，这里很安全，很温暖，肯定没有老虎。你正在刻苦学习，准备应付考试。也可能想学一些比较难的技术，你的老板认为掌握这种技术需要一周时间，最多不超过十天。这就存在一个问题。你的大脑很想给你帮忙。它会努力地把这些显然不太重要的内容赶走，保证这些东西不去侵占本不算充足的脑力资源。这些资源最好还是用来记住确实重要的事情，比如大老虎，再比如火灾险情。如果你曾经只身着短衣裤被大雪围困，这件事肯定不会忘却，你的大脑会记住绝不要让这种情况再发生第二次。

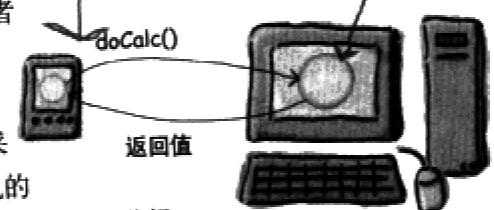
我们没有一种简单的方法来告诉大脑：“嘿，大脑，真是谢谢你了，不过不管这本书多没意思，也不管我对它是多么的无动于衷，但我确实希望你能帮助我把这些东西记下来。”

我们认为"Head First"的读者就是要学习的人

那么，怎么来学习呢？首先，必须了解，然后要保证自己确实不会忘记。这可不是填鸭式的硬塞。根据认知科学、神经生物学和教育心理学的最新研究，学习的途径相当丰富，绝非只是通过书本上的文字。我们很清楚怎么让你的大脑兴奋起来。

下面是一些Head First学习原则：

看得到。与单纯的文字相比，图片更能让人记得住，通过图片，学习效率会更高（对于记忆和传递型的学习，甚至能有多达89%的效率提升）。而且图片更能让人看懂。以往总是把图片放在一页的最下面，甚至放在另外的一页上，与此不同，如果把文字放在与之相关的图片内部，或者在图片的周围写上相关文字，学习者的能力就能得到多至两倍的提高，从而能更好地解决有关的问题。



采用一种针对个人的交谈式风格。最新的研究表明，如果学习过程中采用一种第一人称的交谈方式直接向读者讲述有关内容，而不是用一种干巴巴的语调介绍，学生在学习之后的考试中成绩会提高40%。正确的做法是讲故事，而不是做报告。要用通俗的语言。另外不要太严肃。如果你面对着这样两个人，一个是你在餐会上结识的很有意思的朋友，而另一个学究气十足，喋喋不休地对你说教，在这两个人中，你会更注意哪一个呢？

抽象方法真是简单。这些方法是没有身体的。



让学习的人想得更深。换句话说，除非你很积极地让神经元活动起来，否则你的头脑里什么也不会发生。必须引起读者的好奇，促进、要求并鼓励读者去解决问题、得出结论、产生新的知识。为此，需要提出挑战，留下练习题和拓宽思路的问题，并要求读者完成一些实践活动，让左右脑都开动起来，而且要利用多种思维。

说浴缸是一个（IS-A）浴室合适吗？能不能说浴室是一个浴缸？或者，是不是应该说这是一种"有一个"（HAS-A）关系？



引起读者的注意，而且要让他一直保持注意。我们可能都有这样的体验，"我真的想把这个学会，不过看过一页后就变得昏昏欲睡了"。你的大脑注意的是那些不一般、有意思、有些奇怪、抢眼的、意料之外的东西。学习一项有难度的新技术并不一定枯燥。如果学习过程不乏味,你的大脑很快就能学会。

```
abstract void roam();
```

没有方法体（方法内的定义）！用一个分号结束。

影响读者的情绪。现在我们知道了，记忆能力很大程度上取决于所记的内容对我们的情绪有怎样的影响。如果是你关心的东西，就肯定记得住。如果让你感受到了什么，这些东西就会留在你的脑海中。不过，我们所说的可不是什么关于男孩与狗的伤心故事。这里所说的情绪是惊讶、好奇、觉得有趣。想知道"什么……"，还有就是一种自豪感，如果你解决了一个难题，学会了所有人都觉得很难的东西，或者发现你了解的一些知识竟是那些自以为无所不能的傲慢家伙所不知道的，此时就会有一种自豪感油然而生。



元认知：有关思考的思考

如果你是真的想学，而且想学得更快、更深入，就应该注意你怎样才能集中注意力。考虑自己是怎样思考的，并了解自己的学习方法。

我们中间大多数人长这么大可能都没有上过有关元认知或学习理论的课程。我们想学习，但是很少有人教我们怎么来学习。



不过，这里可以做一个假设，如果你手上有这本书，你想学设计模式，而且可能不想花太多时间。另外，因为你要参加考试，所以需要记住你读到的所有内容。为此必须理解这些内容。想要最大程度地掌握这本书或其他任何一本书中介绍的知识，就要让你的大脑负起责任来，要求它记住这些内容。

怎么做到呢？技巧就在于要让你的大脑认为你在学习的新东西确实很重要，对你的生活有很大影响。就像老虎出现在面前一样。如若不然，你将陷入旷日持久的拉锯战中，虽然你很想记住所学的新内容，但是你的大脑却会竭尽全力地把它们拒之门外。

那么，究竟怎样才能让你的大脑把设计模式看作是一只饥饿的老虎呢？

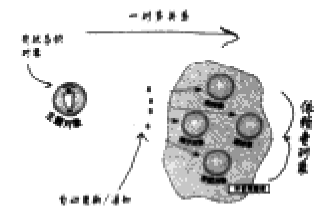
这有两条路：一条比较慢，很乏味；另一条路不仅更快，还更有效。慢方法就是大量地重复。你肯定知道，如果反反复复地看到同一个东西，即使再没有意思，你也能学会并记住它。如果做了足够的重复，你的大脑就会说“尽管看上去这对他来说好像不重要，不过，既然他这样一而再、再而三地看同一个东西，那么我就假定这是很重要的。”

更快的方法是尽一切可能让大脑活动起来，特别是开动大脑来完成不同类型的活动。如何做到这一点呢？上一页列出的学习原则正是一些主要的可取做法，而且经证实，它们确实有助于让你的大脑全力以赴。例如，研究表明，把文字放在所描述图片的中间（而不是放在这一页的别处，比如作为标题，或者放在正文中），这样会让你的大脑更多地考虑这些文字与图片之间有什么关系，而这就会让更多的神经元点火。让更多的神经元点火＝你的大脑更有可能认为这些内容值得注意，而且很可能需要记下来。

交谈式风格也很有帮助，当人们意识到自己在与“别人”交谈，往往会更加关注，这是因为他们总想跟上谈话的思路，并能做出适当的发言。让人惊奇的是，大脑并不关心“交谈”的对方究竟是谁，即使你只是与一本书“交谈”，它也不会不乎！另一方面，如果写作风格很正式，干巴巴的，你的大脑就会觉得像坐在一群人当中被动地听人做报告一样，很没意思，所以不必在意对方说的是什么，甚至可以打瞌睡。

不过，图片和交谈风格还只是开始而已，能做的还有很多。

我们是这么做的：



我们用了很多图，因为你的大脑更能接受看得见的东西，而不是纯文字。对你的大脑而言，一幅图顶得上1024个字。如果既有图片又有文字，我们会把文字放在图片当中，因为文字处在所描述的图片中间时，大脑的工作效率更高，倘偌把这些描述文字作为标题，或者“湮没”在别处的大段文字中，那就达不到这种效果了。

我们采用了重复手法，会用不同的方式，采用不同类型的媒体、运用多种思维手段来介绍同一个东西，目的是让有关内容更有可能储存在你的大脑中，而且能够在多个区中都有容身之地。



模式大师

我们会用你想不到的方式运用概念和图片，因为你的大脑喜欢新鲜玩艺；在提供图和思想时，至少会含着一些情绪因素，因为如果能产生情绪反应，你的大脑就会投入更大的注意。而这会让你感觉到这些东西更有可能要被记住，其实这种感觉可能只是有点幽默，让人奇怪或者比较感兴趣而已。

我们采用了一种针对个人的交谈式风格，因为当你的大脑认为你在参与一个交谈，而不是被动地听一场演示汇报时，它就会更加关注。即使你实际上在读一本书，也就是说在与书“交谈”，而不是真正与人交谈，但这对你的大脑来说并没有什么分别。

在这本书里，我们加入了40多个实践活动，因为与单纯的阅读相比，如果能实际做点什么，你的大脑会更乐于学习，更愿意去记。练习都是我们精心设计的，有一定的难度，但是确实能做出来，因为这是大多数人所希望的。



要点

我们采用了多种学习模式，因为尽管你可能想循序渐进地学习，但是其他人可能希望先对整体有一个全面认识，另外可能还有人只是想看一个代码示例。不过，不管你想怎么学，要是同样的内容能以多种方式来表述，这对每一个人都会有好处。

谜题



这里的内容不只是单单涉及左脑，也不只是让右脑有所动作，我们会让你的左右脑都开动起来，因为你的大脑参与得越多，你就越有可能学会并记住，而且能更长时间地保持注意力。如果只有一大脑在工作，通常意味着另一半有机会休息，这样你就能更有效率地学习更长时间。

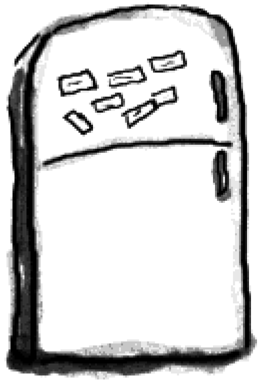
我们会讲故事，留练习，从多种不同的角度来看同一个问题，因为如果要求大脑做一些评价和断，它就能更深入地学习。

你会看到我们给出的一些练习，还要回答一些问题，这些问题往往不是直截了当就能做出回答的，通过克服这些挑战，你就能学得更好，因为让大脑真正做点什么的话，它就更能学会并记住。想想吧，如果只是在健身馆里看着别人流汗，这对于保持你自己的体形肯定不会有什么帮助，正所谓临渊羡鱼，不如退而结网。不过另一方面，我们会竭尽所能不让你钻牛角尖，把劲用错了地方，而是能把功夫用在点子上。也就是说，你不会为搞定一个难懂的例子而耽搁，也不会花太多时间去弄明白一段晦涩难懂而且通篇行话的文字，我们的描述也不会太过简洁而让人无从下手。



我们用了拟人手法。在故事中，在示例中，还有在图中，你都会看到人的出现。这是因为你本身是一个人，不错，这就是原因。如果和人打交道，相对于东西而言，你的大脑会表示出更多的注意。

我们充分利用了80/20方法，我们认为，如果你真的要攻读软件设计博士的话，这本书肯定不会是你唯一的设计模式书，所以我们不打算面面俱到。这里只提供了你真正需要的东西。



可以用下面的方法让你的大脑就范

沿着虚线剪下，贴在冰箱上。

好了，我们该做的已经做了，剩下的就要看你自己的了。这些提示只是个开头：听一听你的大脑是怎么说的，弄清楚对你来说哪些做法可行，哪些做法不能奏效。还可以做些新的尝试。

---

1 慢一点，你理解的越多，需要记的就越少。

不要光是看看而已。停下来，好好想一想。书中提出问题的时候，你不要直接去翻答案。可以假想成真的有人在问你问题。你让大脑想得越深，就越有可能学会并记住。

2 勤做练习，自己记笔记。

我们给你留了练习，但是如果这些练习的解答也由我们一手包办，那和有人替你参加考试有什么区别？不要只是坐在那里看着练习发呆。拿出笔来，写一写、画一画。大量研究都证实，学习过程中如果能实际动动手，将改善你的学习效果。

3 阅读“There are no Dumb Questions”部分。

顾名思义，这些问题可不是可有可无的旁注，它们绝对是核心内容的一部分！千万不要把它们跳过去不看。

4 上床睡觉之前不要再看别的书了,或者至少不再看其他有难度的东西。

学习中有一部分是在你合上书之后完成的（特别是，要把学到的知识长久地记住，这往往无法在看书的过程中做到）。你的大脑也需要有自己的时间来再做一些处理。如果在这段处理时间内你又往大脑里灌输了新的知识，那么你刚学的一些东西就会被丢掉。

5 要喝水，而且要多喝点水。

如果能提供充足的液体，你的大脑才能有最佳表现。如果缺水（可能你觉到口渴之前，就已经缺水了），学习能力就会下降。

6 大声说出来。

说话可以刺激大脑的另一部分。如果你想看懂什么，或者想更牢地记住它，就要大声说出来。更好的办法是，大声地解释给别人听。这样你会学得更快，而且可能会有一些新的认识，而这是以前光看不说的时候未曾发现的。

7 听听你的大脑怎么说。

注意一下你的大脑是不是负荷太重了，如果发现自己开始浮光掠影地翻看，或者刚看的东西就忘记了，这说明你该休息一会儿了。达到某个临界点时，如果还一味地向大脑里塞，这对加快学习速度根本没有帮助，甚至还可能影响正常的学习。

8 要有点感觉！

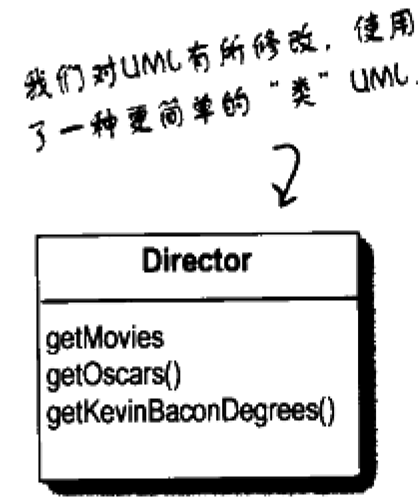
你的大脑需要知道这是很重要的东西。要真正融入到书中的故事里。为书里照片加上你自己的说明。你可能觉得一个笑话很憋脚，不太让人满意，但这总比根本无动于衷要好。

9 设计一些东西！

将学来的知识应用到新项目中，甚至重构旧项目。反正就是尽量应用知识，获取实践经验。你所需要的是一枝铅笔和一个难题，试着应用数个设计模式解决这个难题。

Readme

这是一本体验式学习的书，不是一本参考书。对于学习过程有所阻挠的东西，我们都予以排除。读完第一次之后，你需要从头再读一次，因为本书对读者的背景知识做了一些假设。



我们使用简单的"类"UML图（注意，可不是UML类图，而是指与UML图很相似）。

书中用到了UML，但是我们没有详细介绍UML，而UML也不是本书必备的预备知识。如果你以前没见过UML，也别担心。我们会沿路告诉你一些UML的基本用法。换句话说，你根本不需要同时担心UML和设计模式。我们的图示法是"类"UML图——虽然我们试着用真正的UML，但是基于自私的写作必要，我们终究还是做了一些小改变。

我们没有包含所有的设计模式。

设计模式实在是太多了，GoF的基础模式、Sun的J2EE模式、JSP模式、架构模式、游戏设计模式…… 我们希望这本书的重量能比读者的体重更轻，所以自然不可能涵盖所有的设计模式。我们从GoF模式中，取出更重要的一部分模式，作为本书的焦点，并确保你能够真正地、深入地、彻底地了解如何使用这些模式，以及何时使用这些模式。对于GoF的其他模式，我们也会在附录中概略地介绍。我们相信，读过本书之后，你可以很快地从其他资源中学到本书没有介绍的模式，并且游刃有余。

书里的实践活动不是可有可无的。

这里的练习和实践活动并非可有可无的装饰和摆设；它们也是这本书核心内容的一部分。其中有些练习和活动有助于记忆，有些则能够帮助你理解，还有一些对于如何应用你所学的知识很有帮助。所以，请不要略过这些练习。填字游戏是你唯一可以不理会的部分，但是它们可以帮助大脑回想本章的内容。

当我们提到"组合"（composition）一词，我们指的是OO一般概念中的composition，而不是UML严格定义的composition。

当我们说"一个对象和另一个对象组合在一起"，我们的意思是"有一个"（HAS-A）的关系。在一般的OO概念及GoF的书中，都是采用这样的用法。最近UML对于composition有严谨的定义，如果你是UML专家，你还是可以读这本书，只是要注意到此名词定义上的差异。

我们有意安排了许多重复，这些重复非常重要。

Head First系列图书有一个与众不同的地方，这就是，我们希望你确确实实地掌握这些知识。另外，我们希望在学完这本书之后你能记住学过了什么。尽管重复很有必要，不过，多数参考书都不认为重复和回顾是一个重要的环节，但是在这本书里，你会看到一些概念会一而再、再而三地出现很多次。

代码示例尽可能短小精悍。

有读者告诉我们，如果查了200行代码才能找到要理解的那两行代码，这是很让人郁闷的。这本书里大多数示例往往都开门见山，作为上下文的代码会尽可能地少，这样你就能一目了然地看到哪些东西是需要你学习的。别指望这些代码很健壮，要知道这里的代码甚至是不完整的——毕竟我们的代码是辅助学习之用，所以不见得一定功能完整。

在某些例子中，我们并未将所有需要的package都import进来，但如果你是Java程序员，你应该知道ArrayList类是属于java.util package。如果package不属于J2SE API，我们会特别说明。我们已经将所有的代码都放在网络上，可供下载。网址在：http://wickedlysmart.com/headfirstdesignpatterns/code.html。

为了方便学习与测试程序，我们在书中并没有将我们的类放在package中（换句话说，所有的类都是在Java默认的package中）。我们不建议你在真实世界中也这么做。如果你到我们的网站下载代码，会发现这些类都放在适当的package中。

“Brain Power”习题没有答案。

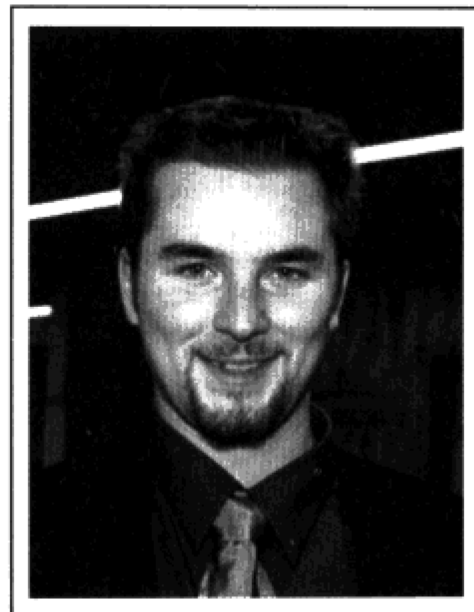
对于某些人来说，“Brain Power”习题没有对的答案；对于另一些人来说，动动脑习题所带来的学习经验在于决定是否你的答案是对的，以及何时你的答案是对的。在某些动动脑习题中，我们会提供暗示，为你指引正确的方向。

技术审校群

Valentin Crettaz

Jef Cumps

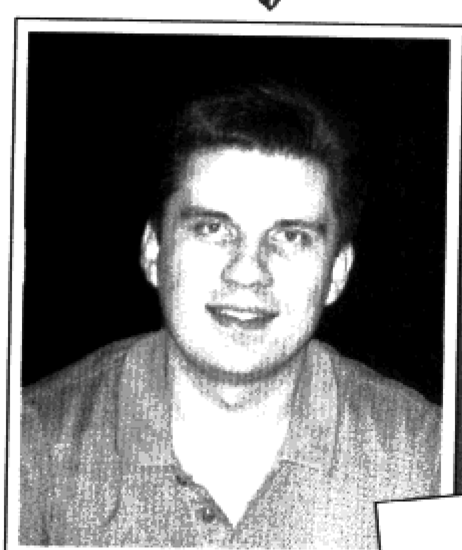




Barney Marispini



Ike Van Atta





本书狂热的审校团队的无畏的队长

Johannes deJong

Jason Menard



Mark Spritzler





Dirk Schreckmann



Philippe Maquet

怀念Philippe Maquet

1960—2004

你那惊人的技术专长、不懈的热忱、为学习者的深思熟虑，将永远激励我们。

我们永远缅怀你。

致谢

致O'Reilly：

在O'Reilly我们对Mike Loukides致以最大的谢意，感谢他开始这一切，并将Head First观念形成一个系列。对Head First幕后的推手Tim O'Reilly致以衷心的感谢。感谢聪明的Head First“系列之母”Kyle Hart，还有摇滚明星Ellie Volkhausen和她灵感十足的封面设计，还有Colleen Gorman的核心编辑。最后，感谢Mike Hendrickson支持这本“设计模式”的书，并建立了整个的团队。

我们大无畏的审校者：

我们特别感谢技术审校的队长Johannes deJong。Johannes，你是我们的英雄。我们深深地感谢Javaranch 审校团队共同管理者的贡献，已故的Philippe Maquet，你以只手照亮了上千开发人员的生活，永远地影响了他们（还有我们）的生活。

Jef Cumps总是能在我们的草拟章节中找出问题，并再三地造成本书巨幅的改变，谢了！Jef!

Valentin Cretazz（专搞AOP的人），他从第一本Head First开始就跟着我们，总是适时地提供我们刚好需要的技术专长，以及他的洞察力。你真行，Valentin。

Head First审阅团队有两位新人，Barney Marispini和Ike Van Atta担任专挑本书毛病的工作，你们两位给我们真正严酷的反馈，谢谢你们的加入。

我们还从Javaranch的主考人/大师——Mark Spritzler、JasonMenard、Dirk Schreckmann、Thomas Paul与Margarita Isaeva等人那里得到杰出的技术帮助。一如平常，要特别感谢javaranch.com Trail的老板Paul Wheaton。

感谢参加“挑选HFDP封面”竞赛的最后决赛入围者。赢家是Si Brewster，他提交了获胜的文字，说服我们选用本书封面的女人。其他入围最后决赛的有：Andrew Esse、Gian Franco Casula、Helen Crosbie、PhoTek、Helen Thomas、Sateesh Kommineni及Jeff Fisher。

还有更多人要感谢*

来自Eric和Elisabeth的感谢

与两位令人惊讶的“导游”——Kathy Sierra和Bert Bates——共同写一本Head First书，是一次放肆的旅行。你们把所有写书的惯例一股脑儿地抛弃，而带我们进入充满说故事、学习理论、认知科学及流行文化的世界，这是读者成为主宰的世界。谢谢两位让我们进入你们的神奇世界，我们希望作这本Head First是正确的。老实说，我们还震惊不已。谢谢你们细心地指导、推动我们向前走，而最主要的就是信任我们（还有你的宝贝）。我们知道，两位都是相当地“鬼灵精”，也都是最时尚的29岁，所以……接下来呢?

要大大地感谢Mike Loukides和Mike Hendrickson。Mike L.从头到尾都伴随着我们。Mike，你有深刻见解的反馈帮助了本书的形成，你的鼓励让我们继续往前走下去。Mike H.，感谢你持续五年来游说我们写一本关于模式的书；我们终于做到了，并且很高兴等到了Head First系列。

特别感谢Erich Gamma，他所做的已经远超过审校本书的责任（甚至在他度假时，都带着本书的草稿）。Erich，你对本书的关注激励了我们，而你彻底的技术审校，无可估量地改善了这本书。同样感谢整个四人组的支持和关注，并且还特别出现在对象村。我们也从Ward Cunningham和模式社群处受惠不少，他们创建了波特兰模式库（Portland Pattern Repository）——我们写本书时不可或缺的资源。

写本技术书需要集结一些人的智慧与力量：Bill Pugh和Ken Arnold在单件模式上，给了我们专业的建议；Joshua Marinacci提供了Swing的技巧和建议；John Brewer的“为什么是鸭子”产生了模拟鸭子的设想（我们很高兴，他也喜欢鸭子）；Dan Friedman激发了小单件的例子；Daniel Steinberg担任我们的技术联络和感情网络；再感谢Apple的James Dempsey，允许我们使用他的MVC歌曲。

最后，私下感谢Javaranch审校团队，为我们做最高级别的校对，以及温馨的支持。还有更多人，没有写在这里……。

来自Kathy和Bert的感谢

我们很想感谢Mike Hendrickson找到Eric和Elisabeth……但是不能。因为这两人，我们发现（令我们恐怖的是），已经不只有我们可以写Head First的书了。不过，如果读者想要相信，在书里所有的“酷事”都是Kathy和Bert的作为，那么，“我们”是谁，可以让他们循规蹈矩?

* 之所以要感谢这么多人，是因为我发现了这样一条定律，书中致谢部分里提到的每个人都至少会买一本书，可能还会买好几本书,给亲戚和周围的所有人都送上一本,如果你希望我们在下一本书的致谢里提到你,而且你们家族的人很多的话,可以写信给我们。

目录（概览）

	引子	xxvii
1	欢迎来到设计模式世界：设计模式入门	1
2	让你的对象知悉现况：观察者模式	37
3	装饰对象：装饰者模式	79
4	烘烤OO的精华：工厂模式	109
5	独一无二的对象：单件模式	169
6	封装调用：命令模式	191
7	随遇而安：适配器与外观模式	235
8	封装算法：模板方法模式	275
9	管理良好的集合：迭代器与组合模式	315
10	事物的状态：状态模式	385
11	控制对象访问：代理模式	429
12	模式中的模式：复合模式	499
13	真实世界中的模式：与设计模式相处	577
14	附录A：剩下的模式	611

目录（真正的目录）

引子

让你的大脑来学设计模式。你想学些东西，但是你的大脑却在帮倒忙，不让你记住这些东西。你的大脑在想，“还是把空间留给更重要的事情吧，比方说要躲避的野兽，还有，光着身子滑雪不太好吧。”那么你该如何骗过大脑，让它认为要是不知道设计模式你就活不下去了？

谁适合读这本书？	xxviii
我们知道你的大脑在想什么	xxix
元认知	xxxi
让你的大脑就范	xxxiii
技术审校	xxxvi
致谢	xxxvii

1

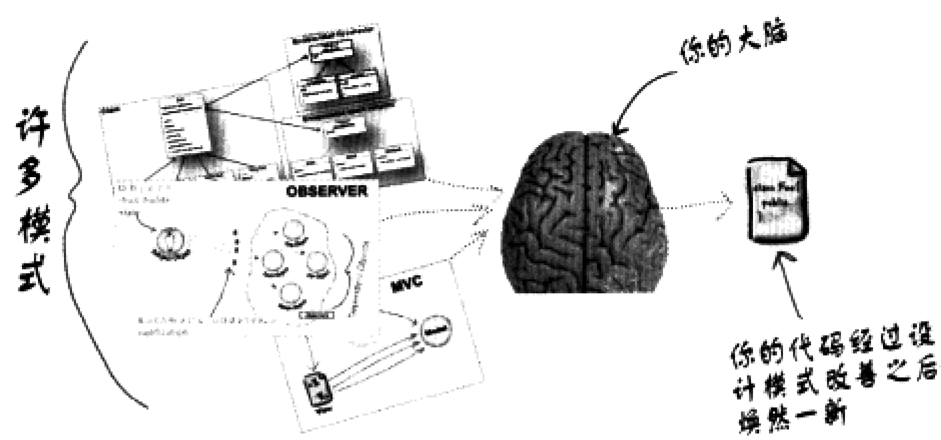
设计模式入门

欢迎来到设计模式世界

有些人已经解决你的问题了。在本章，你将学到为何（以及如何）利用其他开发人员的智慧与经验。他们遭遇过相同的问题，也顺利地解决过这些问题。本章结束前，我们会先看看设计模式的用途与优点，再看一些关键的OO设计原则，并通过一个实例来了解模式是如何运作的。使用模式最好的方式是：“把模式装进脑子里，然后在你的设计和已有的应用中，寻找何处可以使用它们。”以往是代码复用，现在是经验复用。

模拟鸭子应用	2
Joe想到继承	5
利用接口如何？	6
软件开发的不变真理	8
分开变化和不变部分	10
设计鸭子的行为	11
测试鸭子的代码	18
动态地设置行为	20
封装行为的大局观	22
“有一个”比“是一个”更好	23
策略模式	24
共享模式词汇的威力	28
我如何使用设计模式？	29
设计箱内的工具	32
习题解答	34

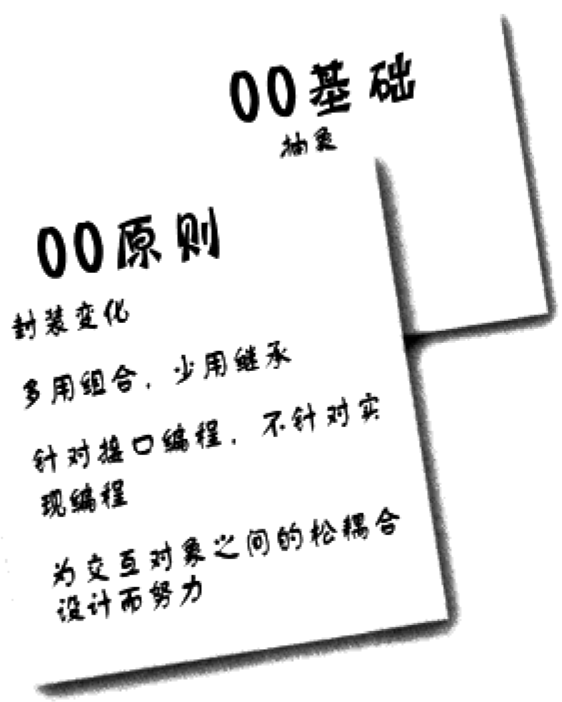




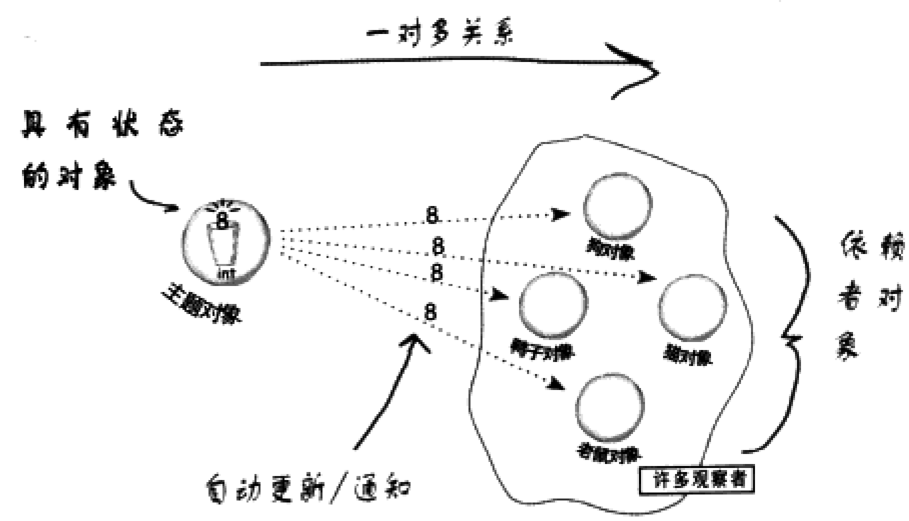
2 观察者模式

让你的对象知悉现况

有趣的事情发生时，可千万别错过了！ 有一个模式可以帮你的对象知悉现况，不会错过该对象感兴趣的事。对象甚至在运行时可决定是否要继续被通知。观察者模式是JDK中使用最多的模式之一，非常有用。我们也会一并介绍一对多关系，以及松耦合（对，没错。我们说耦合）。有了观察者，你将会消息灵通。



气象观测站	39
认识观察者模式	44
出版者+订阅者=观察者模式	45
五分钟短剧：观察主题	48
定义观察者模式	51
松耦合的威力	53
设计气象站	56
实现气象站	57
使用Java内建的观察者模式	64
java.util.Observable的黑暗面	71
设计箱内的工具	74
习题解答	78



3 装饰者模式

装饰对象

本章可以称为“给爱用继承的人一个全新的设计眼界”。我们即将再度探讨典型的继承滥用问题。你将在本章学到如何使用对象组合的方式，做到在运行时装饰类。为什么呢？一旦你熟悉了装饰的技巧，你将能够在不修改任何底层类代码的情况下，给你的（或别人的）对象赋予新的职责。

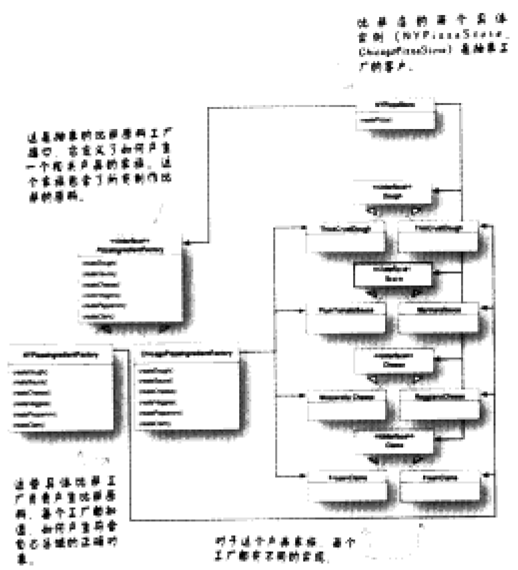


欢迎来到星巴兹咖啡 80
开放－关闭原则 86
认识装饰者模式 88
以装饰者构造饮料订单 89
定义装饰者模式 91
装饰饮料 92
写下星巴兹的代码 95
真实世界的装饰者：Java I/O 100
编写自己的Java I/O装饰者 102
设计箱内的工具 105
习题解答 106

工厂模式

4 烘烤OO的精华

准备好开始烘烤某些松耦合的OO设计。除了使用new操作符之外，还有更多制造对象的方法。你将了解到实例化这个活动不应该总是公开地进行，也会认识到初始化经常造成“耦合”问题。你不希望这样，对吧？读下去，你将了解工厂模式如何从复杂的依赖中帮你脱困。



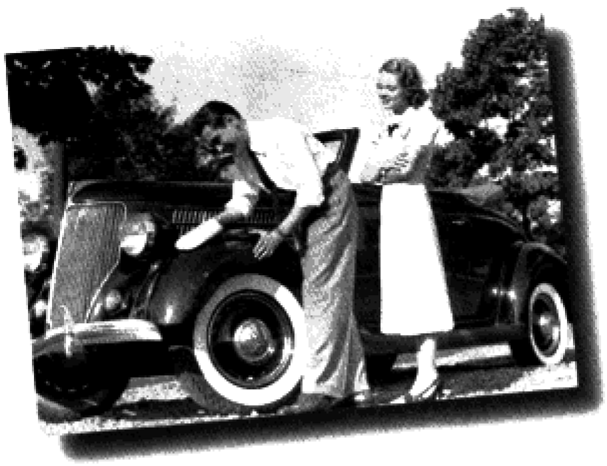


当看到“new”，就会想到“具体”	110
对象村比萨	112
封装创建对象的代码	114
建立一个简单比萨工厂	115
定义简单工厂	117
给比萨店使用的框架	120
允许子类做决定	121
让我们开一家比萨店吧	123
声明一个工厂方法	125
认识工厂方法模式	131
平行的类层级	132
定义工厂方法模式	134
一个很依赖的比萨店	137
看看对象依赖	138
依赖倒置原则	139
再回到比萨店……	144
原料家族	145
建造原料工厂	146
看看抽象工厂	153
幕后花絮	154
定义抽象工厂模式	156
比较工厂方法和抽象工厂	160
设计箱内的工具	162
习题解答	164

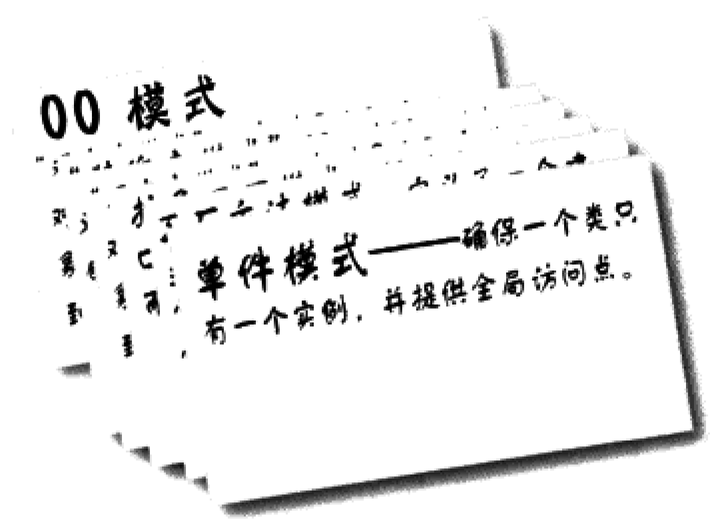
5 单件模式

独一无二的对象

单件模式：用来创建独一无二的，只能有一个实例的对象的入场券。告诉你一个好消息，单件模式的类图可以说是所有模式的类图中最简单的，事实上，它的类图上只有一个类！但是，可不要兴奋过头，尽管从类设计的视角来说很简单，但是实现上还是会遇到相当多的波折。所以，系好安全带，出发了！



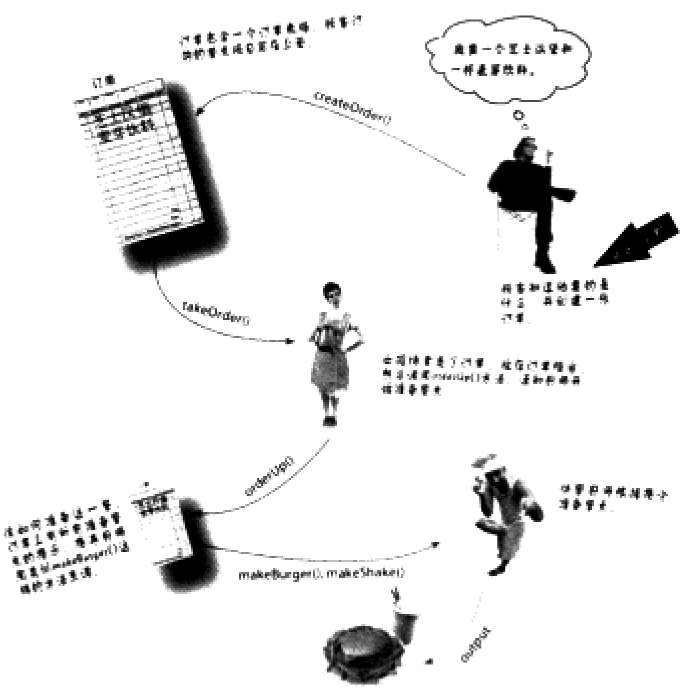
独一无二	170
小小单件	171
剖析经典的单件模式实现	173
单件的告白	174
巧克力工厂	175
定义单件模式	177
Hershey ~~Houston,~~ 我们遇到麻烦了……	178
化身为JVM	179
处理多线程	180
单件Q&A	184
设计箱内的工具	186
习题解答	188



6 命令模式

封装调用

在本章，我们将把封装带到一个全新的境界：把方法调用封装起来。没错，通过封装方法调用，我们可以把运算块包装成形。所以调用此运算的对象不需要关心事情是如何进行的，只要知道如何使用包装成形的方法来完成它就可以。通过封装方法调用，也可以做一些很聪明的事情，例如记录日志，或者重复使用这些封装来实现撤销。

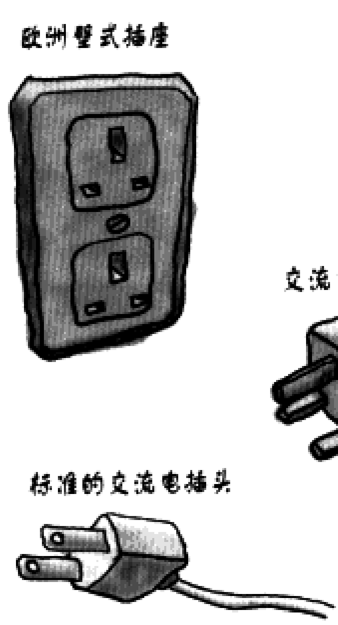


巴斯特家电自动化公司 192
遥控器 193
看一下厂商的类 194
同时，回到餐厅…… 197
研究餐厅的交互 198
对象村餐厅的角色和职责 199
从餐厅到命令模式 201
第一个命令对象 203
定义命令模式 206
命令模式与遥控器 208
实现遥控器 210
逐步测试遥控器 212
写文档的时刻到了 215
使用状态实现撤销 220
每个遥控器都需要Party模式！ 224
使用宏命令 225
命令模式的更多用途：队列请求 228
命令模式的更多用途：日志请求 229
设计箱内的工具 230
习题解答 232

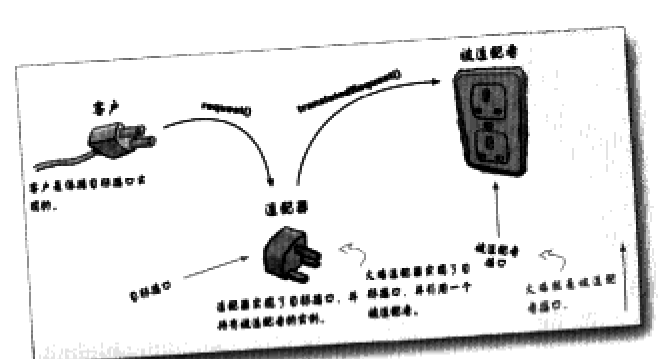
7 适配器模式与外观模式

随遇而安

在本章，我们将要进行一项任务，其不可能的程度，简直就像是将一个方块放进一个圆洞中。听起来不可能？有了设计模式，就有可能。还记得装饰者模式吗？我们将对象包装起来，赋予它们新的职责。而现在则是以不同目的，包装某些对象：让它们的接口看起来不像自己而像是别的东西。为何要这样做？因为这样就可以在设计中，将类的接口转换成想要的接口，以便实现不同的接口。不仅如此，我们还要探讨另一个模式，将对象包装起来以简化其接口。



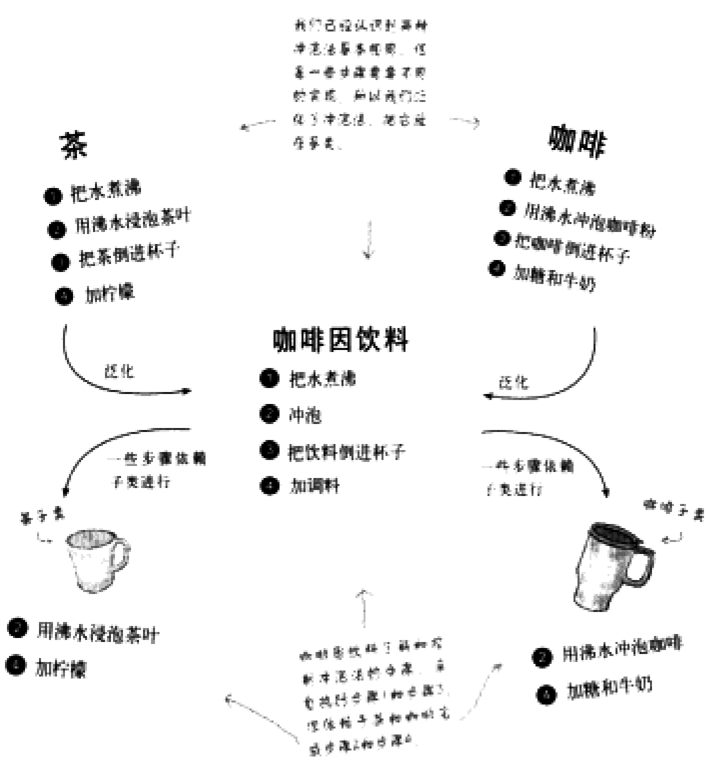
我们周围的适配器 236
面向对象适配器 237
适配器模式解析 241
定义适配器模式 243
对象和类的适配器 244
今夜话题：对象适配器和类适配器 247
真实世界的适配器 248
将枚举适配到迭代器 249
今夜话题：装饰者模式和适配器模式 252
甜蜜的家庭影院 255
灯光、相机、外观！ 258
构造家庭影院外观 261
定义外观模式 264
“最少知识”原则 265
设计箱内的工具 270
习题解答 272



8 模板方法模式

封装算法

直到目前，我们的议题都绕着封装转；我们已经封装了对象创建、方法调用、复杂接口、鸭子、比萨……接下来呢？我们将要深入封装算法块，好让子类可以在任何时候都可以将自己挂接进运算里。我们甚至会在本章学到一个受到好莱坞影响而启发的设计原则。



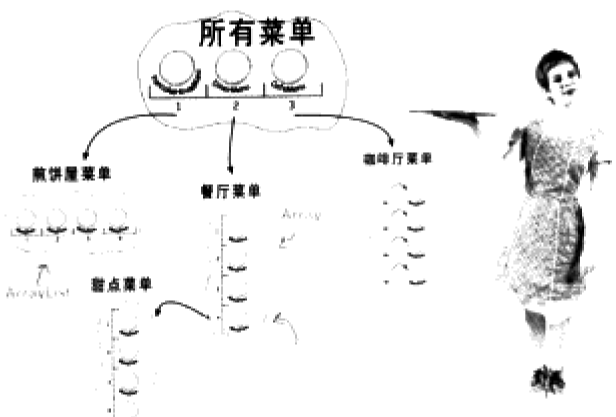
快速搞定几个咖啡和茶的类	277
抽取咖啡和茶	280
更进一步的设计……	281
抽象prepareRecipe()	282
我们做了什么？	285
认识模板方法	286
走，泡茶去	287
模板方法带给我们什么？	288
定义模板方法模式	289
再靠近一点	290
对模板方法进行挂钩……	292
使用钩子	293
咖啡？茶？执行测试程序	294
好莱坞原则	296
好莱坞原则和模板方法	297
荒野中的模板方法	299
用模板方法排序	300
来排序鸭子吧……	301
比较鸭子	302
观察鸭子排序的内部运作	304
写一个Swing的窗口程序	306
Applet	307
今夜话题：模板方法和策略	308
设计箱内的工具	311
习题解答	312

9

迭代器与组合模式

管理良好的集合

有许多种方法可以把对象堆起来成为一个集合。你可以把它们放进数组、堆栈、列表或者是散列表（Hashtable）中，这是你的自由。每一种都有它自己的优点和适合的使用时机，但总有一个时候，你的客户想要遍历这些对象，而当他这么做时，你打算让客户看到你的实现吗？我们当然希望最好不要！这太不专业了。没关系，不要为你的工作担心，你将在本章中学习如何能让客户遍历你的对象而又无法窥视你存储对象的方式；也将学习如何创建一些对象超集合（super collection），能够一口气就跳过某些让人望而生畏的数据结构。你还将学到一些关于对象职责的知识。



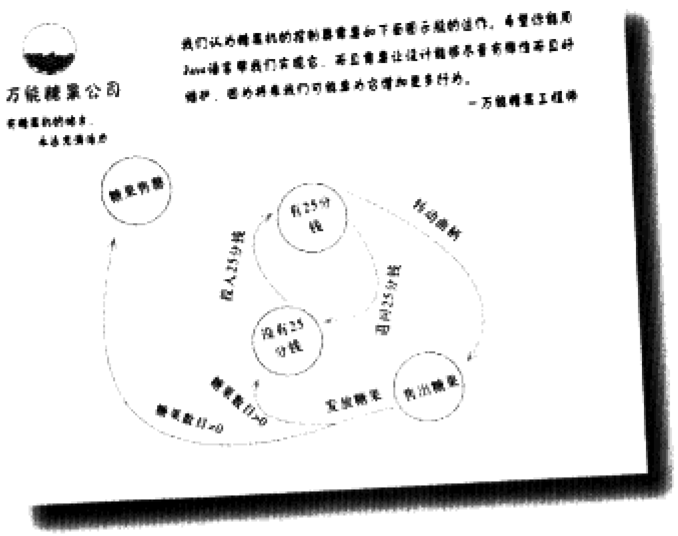
对象村餐厅和对象村煎饼屋合并了	316
比较菜单的实现	318
可以封装遍历吗？	323
认识迭代器模式	325
在餐厅菜单中加入一个迭代器	326
鸟瞰目前的设计	331
利用java.util.Iterator来清理	333
这为我们带来什么好处？	335
定义迭代器模式	336
单一责任	339
迭代器与集合	348
Java 5 的迭代器和集合	349
正当我们认为这很安全的时候……	353
定义组合模式	356
利用组合设计菜单	359
实现组合菜单	362
闪回到迭代器	368
空迭代器	372
迭代器和组合凑在一起的魔力……	374
设计箱内的工具	380
习题解答	381

10 状态模式

事物的状态

基本常识：策略模式和状态模式是双胞胎，在出生时才分开。

你已经知道了，策略模式是围绕可以互换的算法来创建成功业务的。然而，状态走的是更崇高的路，它通过改变对象内部的状态来帮助对象控制自己的行为。它常常告诉它的对象客户“跟着我念：我很棒，我很聪明，我最优秀了……”



如何实现状态？（办公室隔间对话） 387
状态机101 388
状态机代码的第一个版本 390
该来的躲不掉……变更请求！ 394
混乱的状态…… 396
定义状态接口和类 399
实现我们的状态类 401
重新改造糖果机 402
定义状态模式 410
状态 vs.策略模式 411
精神检查…… 417
我们差点儿忘了！ 420
设计箱内的工具 423
习题解答 424



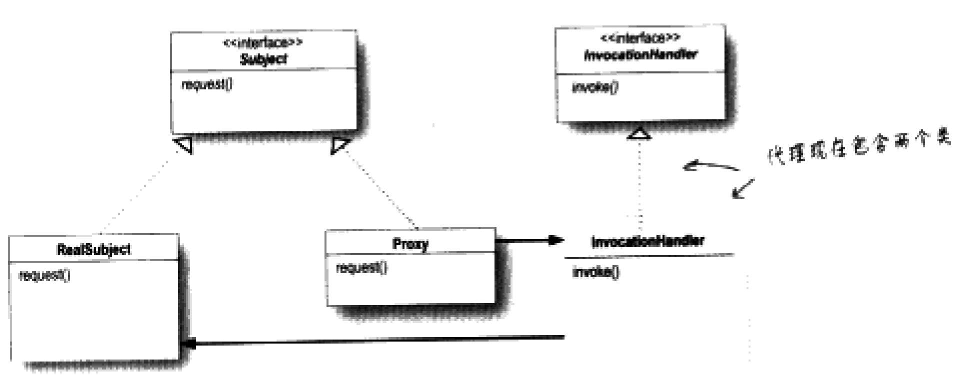
11 代理模式

控制对象访问

玩过扮白脸、扮黑脸的游戏吗？你是一个白脸，提供很好且很友善的服务，但是你不希望每个人都叫你做事，所以找了黑脸控制对你的访问。这就是代理要做的：控制和管理访问。就像你将看到的，代理的方式有许多种。代理以通过Internet为它们的代理对象搬运的整个方法调用而出名，它也可以代替某些懒惰的对象做一些事情。



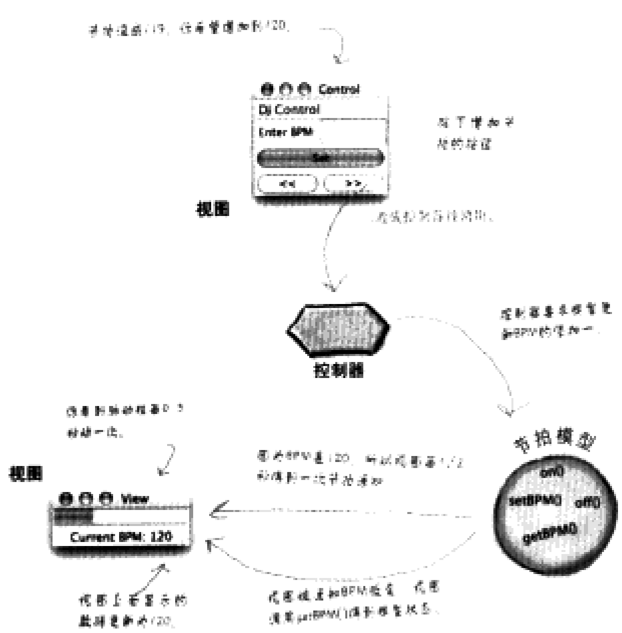
监控糖果机	430
远程代理的角色	434
RMI浏览	437
GumballMachine远程代理	450
代理幕后花絮	458
定义代理模式	460
准备虚拟代理	462
设计CD封面虚拟代理	464
虚拟代理的幕后花絮	470
使用Java API的代理	474
五分钟短剧：保护主题	478
创建动态代理	479
代理动物园	488
设计箱内的工具	491
习题解答	492



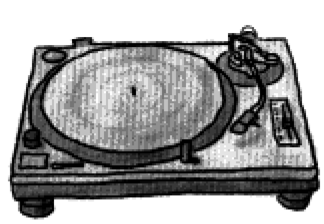
12 复合模式

模式中的模式

谁料得到模式居然可以携手合作？你已经见识过围炉夜话的火爆场面（幸好，出版社事先请我们删除“死神来访”模式的篇章，好让本书不需附上“12岁以下读者必须家长陪同阅读”的警告标语，所以你没见识到闹出人命的那一集围炉夜话），谁料得到模式居然可以携手合作？这实在是太意外了。信不信由你，有一些威力强大的OO设计同时使用多个设计模式。准备让你的模式技巧进入下一个层次，现在是复合模式的时间。



复合模式 500
与鸭子重聚 501
加入一个适配器 504
加入一个装饰者 506
加入一个工厂 508
加入一个组合和一个迭代器 513
加入一个观察者 516
模式概览 523
鸭瞰：类图 524
模型－视图－控制器之歌 526
设计模式是MVC的钥匙 528
戴着模式的有色眼镜看MVC 532
利用MVC控制节拍…… 534
模型 537
视图 539
控制器 542
探索策略 545
适配模型 546
现在我们准备写HeartController 547
MVC与Web 549
设计模式和Model 2 557
设计箱内的工具 560
习题解答 561

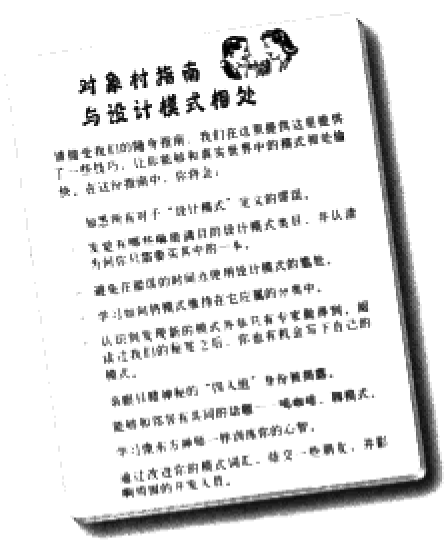


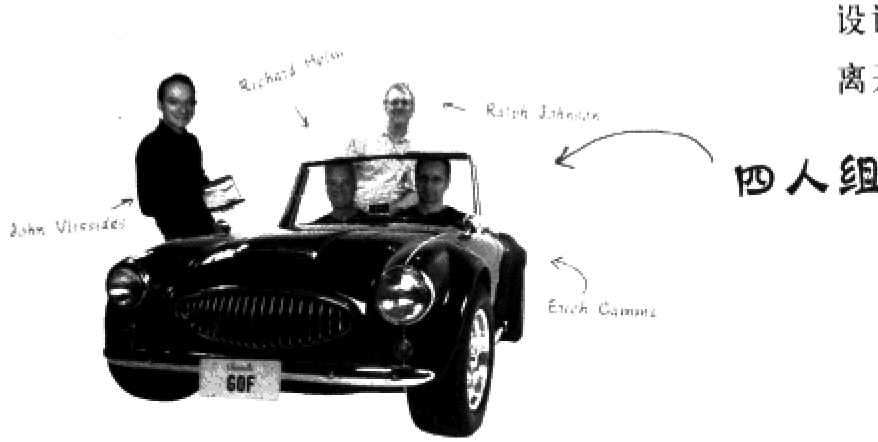
13 与设计模式相处

真实世界中的模式

现在你已经准备好迎接一个充满设计模式的崭新世界。但是，在你打开所有的机会大门之前，我们需要告诉你一些即将在真实世界中遇到的细节——没错，外面的世界比对象村来得复杂。来吧！从下页开始，我们会指引你的方向……

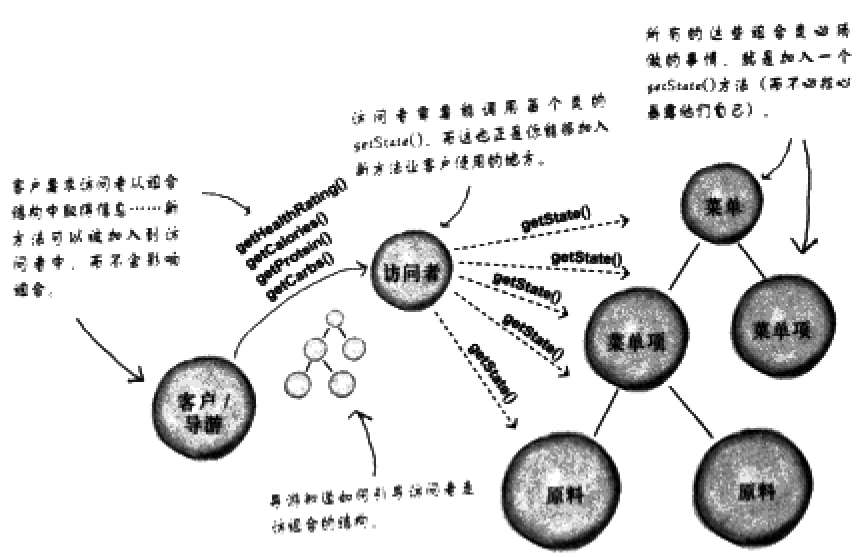
对象村指南	578
定义设计模式	579
更近地观察设计模式的定义	581
愿力与你同在	582
模式类目	583
如何创建模式	586
想当一个设计模式作家吗？	587
组织设计模式	589
用模式思考	594
使用模式的心智	597
别忘了共享词汇的威力	599
共享词汇的五种方式	600
和四人组一同巡游对象村	601
你的旅途刚刚开始……	602
其他设计模式资源	603
模式动物园	604
以反模式歼灭恶势力	606
设计箱内的工具	608
离开对象村……	609





附录A：剩下的模式

并非每个人都广受欢迎。过去10年来，事情改变了许多。自从《设计模式：可复用面向对象软件的基础》一书出版之后，开发人员就开始大量地采用这些模式。我们在此附录中所介绍的模式，都是成熟、典型、正式的四人组模式，只不过可能不像前面章节所探索的模式那么经常地被使用。但是这些模式本身也有相当可取之处，而如果你遇到了合适的情形，也应当毫不犹豫地采用它们。我们在此的目标，是希望能够让你通盘了解这些模式的意义。



桥接	612
生成器	614
责任链	616
蝇量	618
解释器	620
中介者	622
备忘录	624
原型	626
访问者	628

i 索引 631

1 设计模式入门

欢迎来到设计模式世界

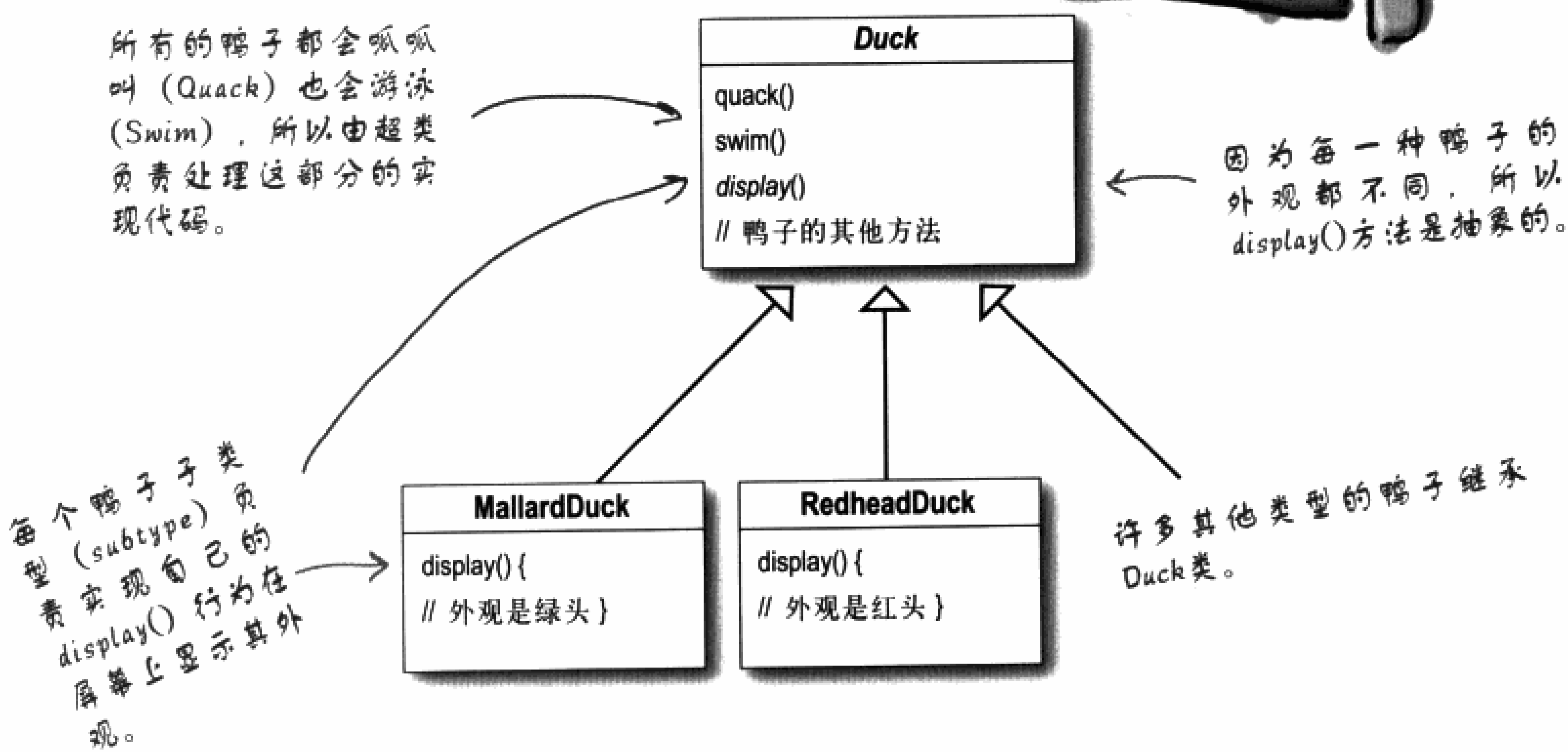


有些人已经解决你的问题了。 在本章，你将学到为何（以及如何）利用其他开发人员的经验与智慧。他们遭遇过相同的问题，也顺利地解决过这些问题。本章结束前，我们会看看设计模式的用途与优点，再看一些关键的OO设计原则，并通过一个实例来了解模式是如何运作。使用模式最好的方式是："把模式装进脑子里，然后在你的设计和已有的应用中，寻找何处可以使用它们。"以往是代码复用，现在是经验复用。

先从简单的模拟鸭子应用做起

Joe上班的公司做了一套相当成功的模拟鸭子游戏：SimUDuck。游戏中会出现各种鸭子，一边游泳戏水，一边呱呱叫。此系统的内部设计使用了标准的OO技术，设计了一个鸭子超类（Superclass），并让各种鸭子继承此超类。

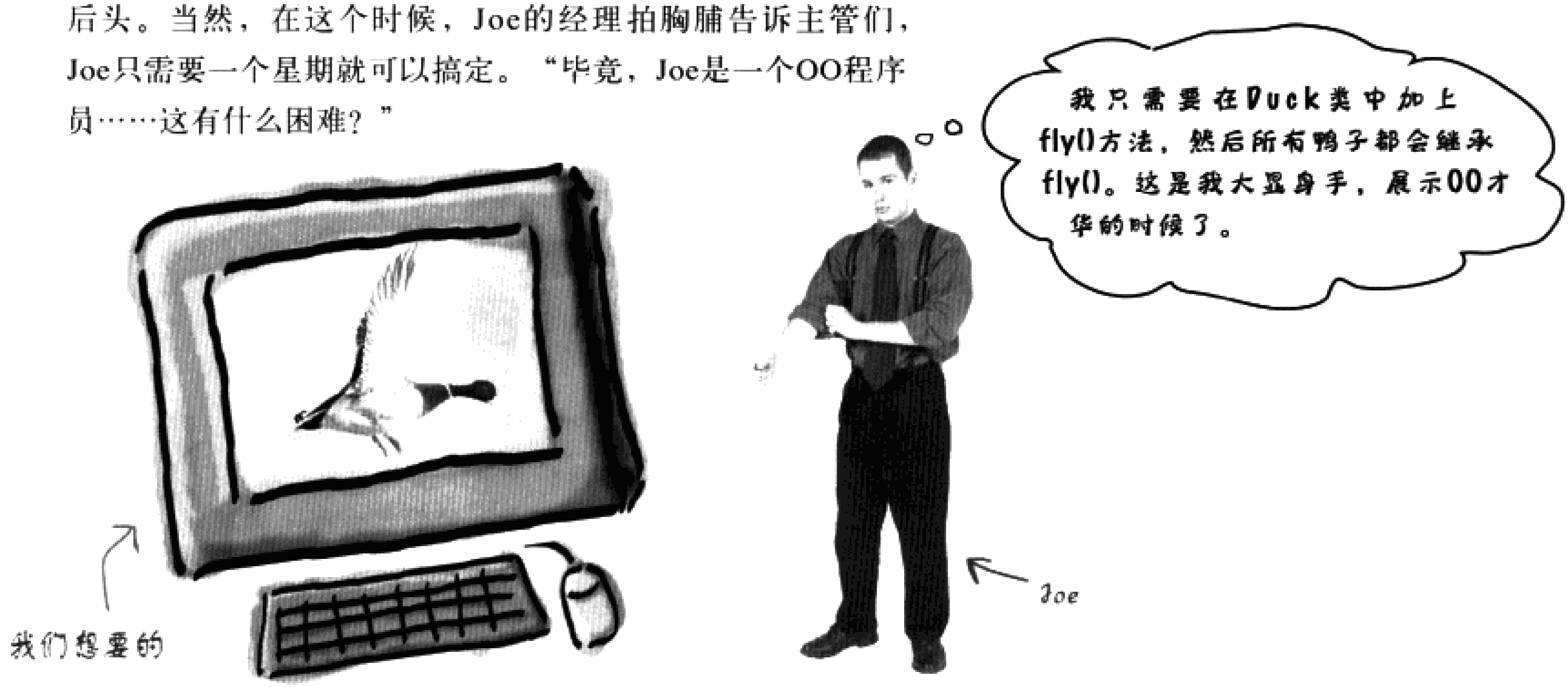


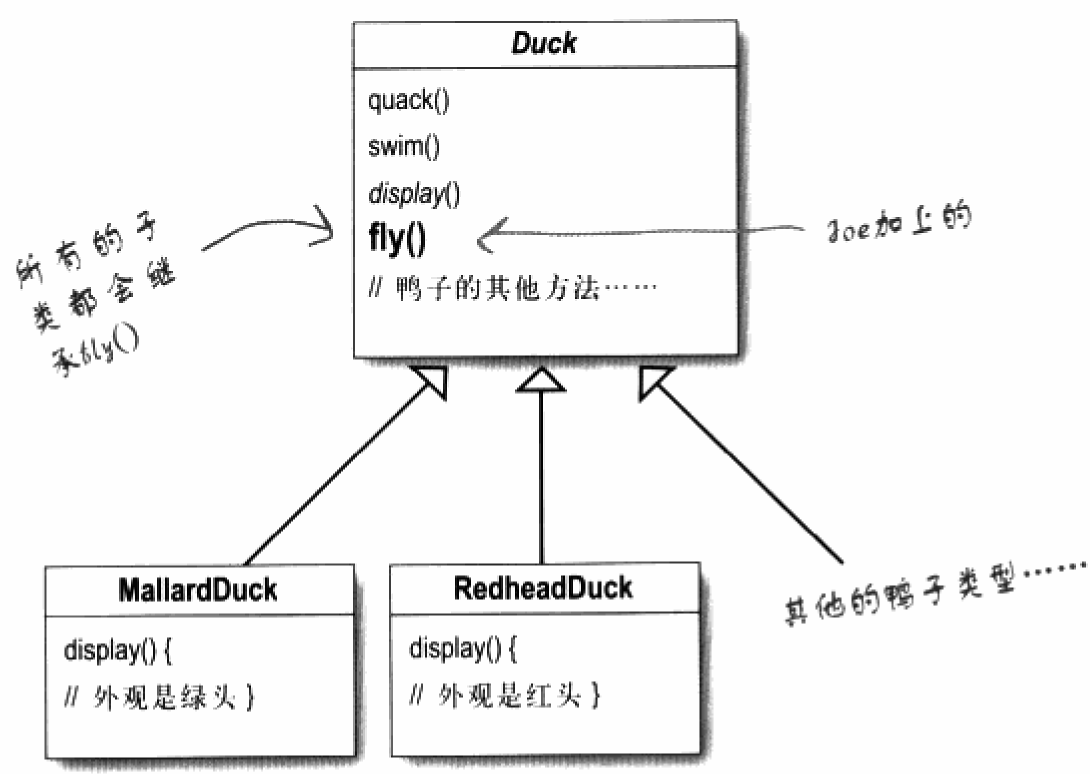


去年，公司的竞争压力加剧。在为期一周的高尔夫假期兼头脑风暴会议之后，公司主管认为该是创新的时候了，他们需要在“下周”毛伊岛股东会议上展示一些“真正”让人印象深刻的东西来振奋人心。

现在我们得让鸭子能飞

主管们确定，此模拟程序需要会飞的鸭子来将竞争者抛在后头。当然，在这个时候，Joe的经理拍胸脯告诉主管们，Joe只需要一个星期就可以搞定。“毕竟，Joe是一个OO程序员……这有什么困难？”





但是，可怕的问题发生了……

Joe，我正在股东会议上，刚刚看了一下展示，有很多"橡皮鸭子"在屏幕上飞来飞去，这是你在开玩笑吗？你可能要开始去逛逛Monster.com（编注：美国最大的求职网站）了……





怎么回事？

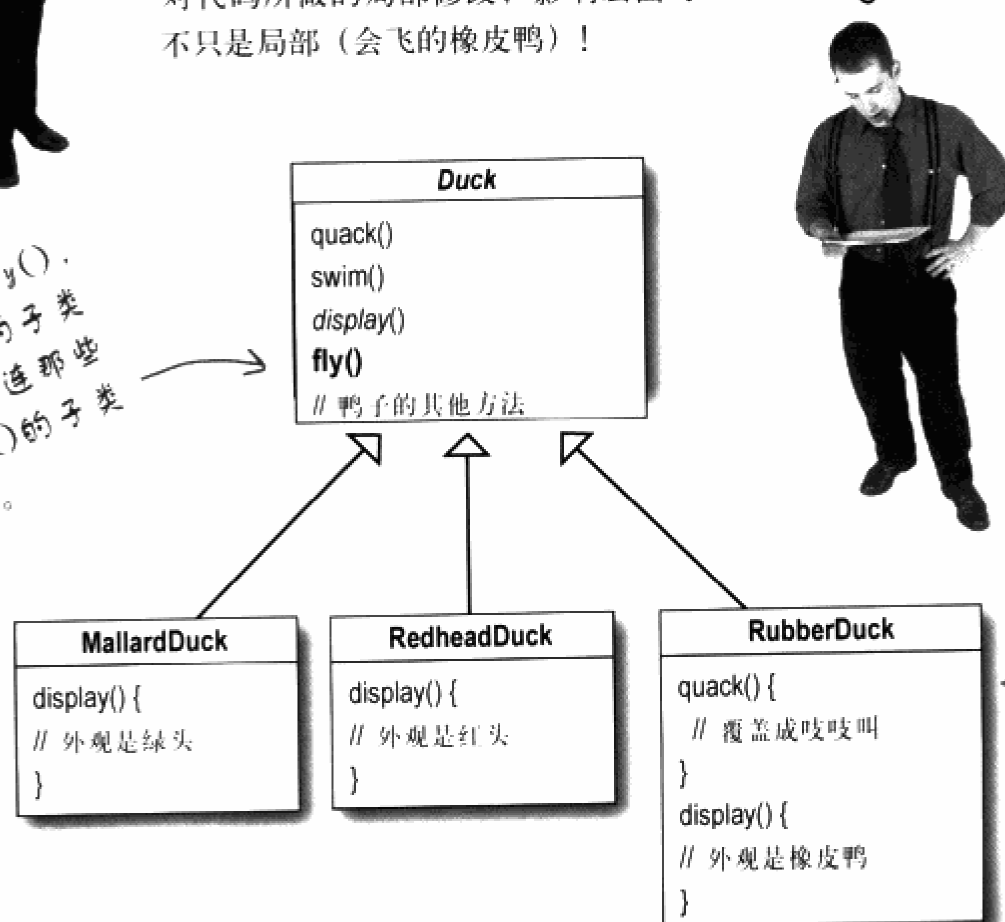
Joe忽略了一件事：并非Duck所有的子类都会飞。Joe在Duck超类中加上新的行为，会使得某些并不适合该行为的子类也具有该行为。现在可好了！SimUDuck程序中有了一个无生命的会飞的东西。

对代码所做的局部修改，影响层面可不只是局部（会飞的橡皮鸭）！

好吧！我承认设计中有一点小疏失。但是，他们怎么不干脆把这当成一种"特色"，其实还挺有趣的呀……

他体会到了一件事：当涉及"维护"时，为了"复用"（reuse）目的而使用继承，结局并不完美。

在超类中加上fly()，就会导致所有的子类都具备fly()，连那些不该具备fly()的子类也无法免除。



橡皮鸭子不会呱呱叫，所以把quack()的定义覆盖成"吱吱叫"（squeak）。

Joe想到继承

我可以把橡皮鸭类中的fly()方法覆盖掉，就好像覆盖quack()的做法一样……



RubberDuck

```
quack() {// 吱吱叫}
display() {// 橡皮鸭 }
fly() {
  // 覆盖，变成什么事都不做
}
```

可是，如果以后我加入诱饵鸭（DecoyDuck），又会如何？诱饵鸭是木头假鸭，不会飞也不会叫……



DecoyDuck

```
quack() {
  // 覆盖，变成什么事都不做
}

display() {// 诱饵鸭}

fly() {
  // 覆盖，变成什么事都不做
}
```

这是继承层次中的另一个类。注意，诱饵鸭既不会飞也不会叫，可是橡皮鸭不会飞但会叫。

Sharpen your pencil

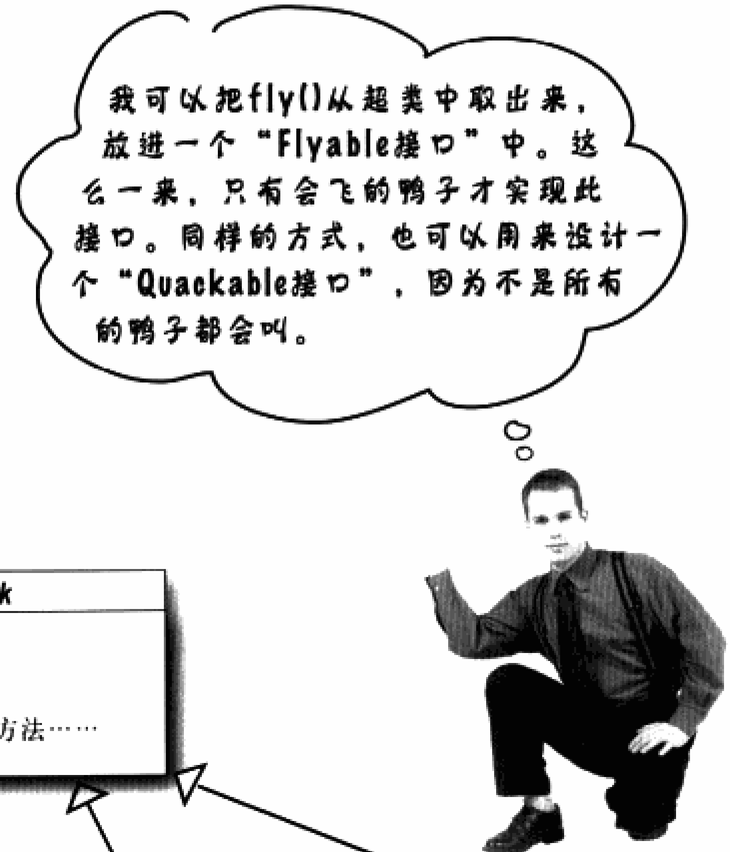
利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

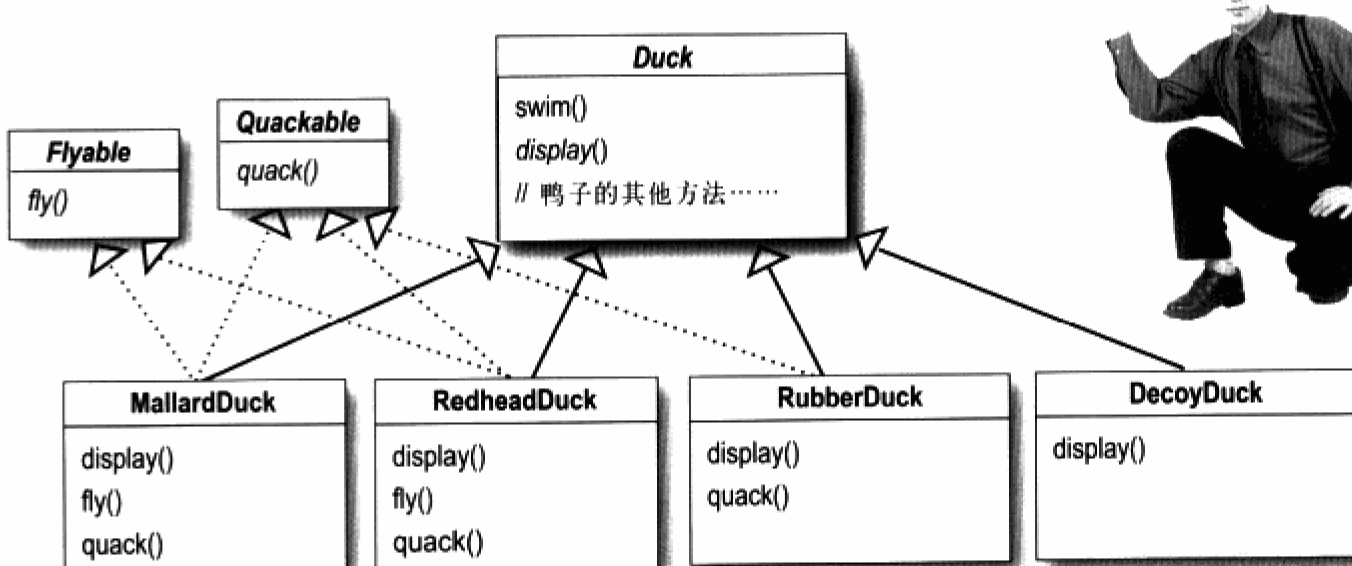
- ☐ A. 代码在多个子类中重复。
- ☐ B. 运行时的行为不容易改变。
- ☐ C. 我们不能让鸭子跳舞。
- ☐ D. 很难知道所有鸭子的全部行为。
- ☐ E. 鸭子不能同时又飞又叫。
- ☐ F. 改变会牵一发动全身，造成其他鸭子不想要的改变。

利用接口如何？

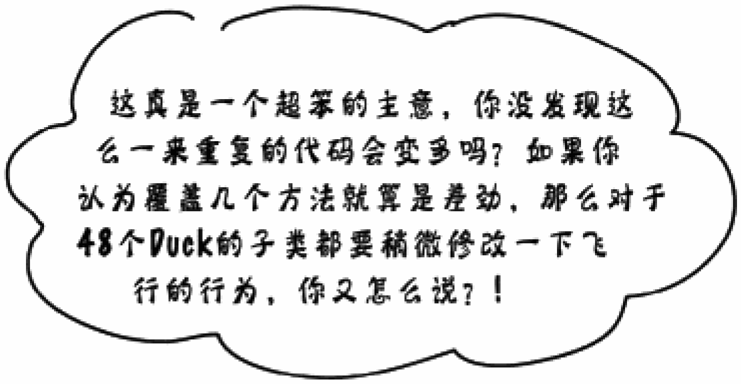
Joe认识到继承可能不是答案，因为他刚刚拿到来自主管的备忘录，希望以后每六个月更新产品（至于更新的方法，他们还没想到）。Joe知道规格会常常改变，每当有新的鸭子子类出现，他就要被迫检查并可能需要覆盖fly()和quark()……这简直是无穷无尽的噩梦。

所以，他需要一个更清晰的方法，让“某些”（而不是全部）鸭子类型可飞或可叫。





你觉得这个设计如何？





如果你是Joe，你要怎么办？

我们知道，并非“所有”的子类都具有飞行和呱呱叫的行为，所以继承并不是适当的解决方式。虽然Flyable与Quackable可以解决“一部分”问题（不会再有会飞的橡皮鸭），但是却造成代码无法复用，这只能算是从一个恶梦跳进另一个恶梦。甚至，在会飞的鸭子中，飞行的动作可能还有多种变化……

此时，你可能正期盼着设计模式能骑着白马来解救你离开苦难的一天。但是，如果直接告诉你答案，这有什么乐趣？我们会用老方法找出一个解决之道：“采用良好的OO软件设计原则”。



软件开发的一个不变真理

好吧！在软件开发上，有什么是你可以深信不疑的？

不管你在何处工作，构建些什么，用何种编程语言，在软件开发上，一直伴随你的那个不变真理是什么？

CHANGE

（用镜子来看答案）

不管当初软件设计得多好，一段时间之后，总是需要成长与改变，否则软件就会“死亡”。

Sharpen your pencil

驱动改变的因素很多。找出你的应用中需要改变代码的原因，一一列出来。（我们写下了一些我们的原因，给你起个头。）

我们的顾客或用户需要别的东西，或者想要新功能。

我的公司决定采用别的数据库产品，又从另一家厂商买了数据，这造成数据格式不兼容。唉！

把问题归零……

现在我们知道使用继承并不能很好地解决问题，因为鸭子的行为在子类里不断地改变，并且让所有的子类都有这些行为是不恰当的。Flyable与Quackable接口一开始似乎还挺不错，解决了问题（只有会飞的鸭子才继承Flyable），但是Java接口不具有实现代码，所以继承接口无法达到代码的复用。这意味着：无论何时你需要修改某个行为，你必须得往下追踪并在每一个定义此行为的类中修改它，一不小心，可能会造成新的错误！

幸运的是，有一个设计原则，恰好适用于此状况。

设计原则

找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。

这是我们的第一个设计原则，以后还有更多原则会陆续在本书中出现。

换句话说，如果每次新的需求一来，都会使某方面的代码发生变化，那么你就可以确定，这部分的代码需要被抽出来，和其他稳定的代码有所区分。

下面是这个原则的另一种思考方式：“把会变化的部分取出并封装起来，以便以后可以轻易地改动或扩充此部分，而不影响不需要变化的其他部分”。

这样的概念很简单，几乎是每个设计模式背后的精神所在。所有的模式都提供了一套方法让“系统中的某部分改变不会影响其他部分”。

好，该是把鸭子的行为从Duck类中取出的时候了！

把会变化的部分取出并“封装”起来，好让其他部分不会受到影响。

结果如何？代码变化引起的不经意后果变少，系统变得更有弹性。

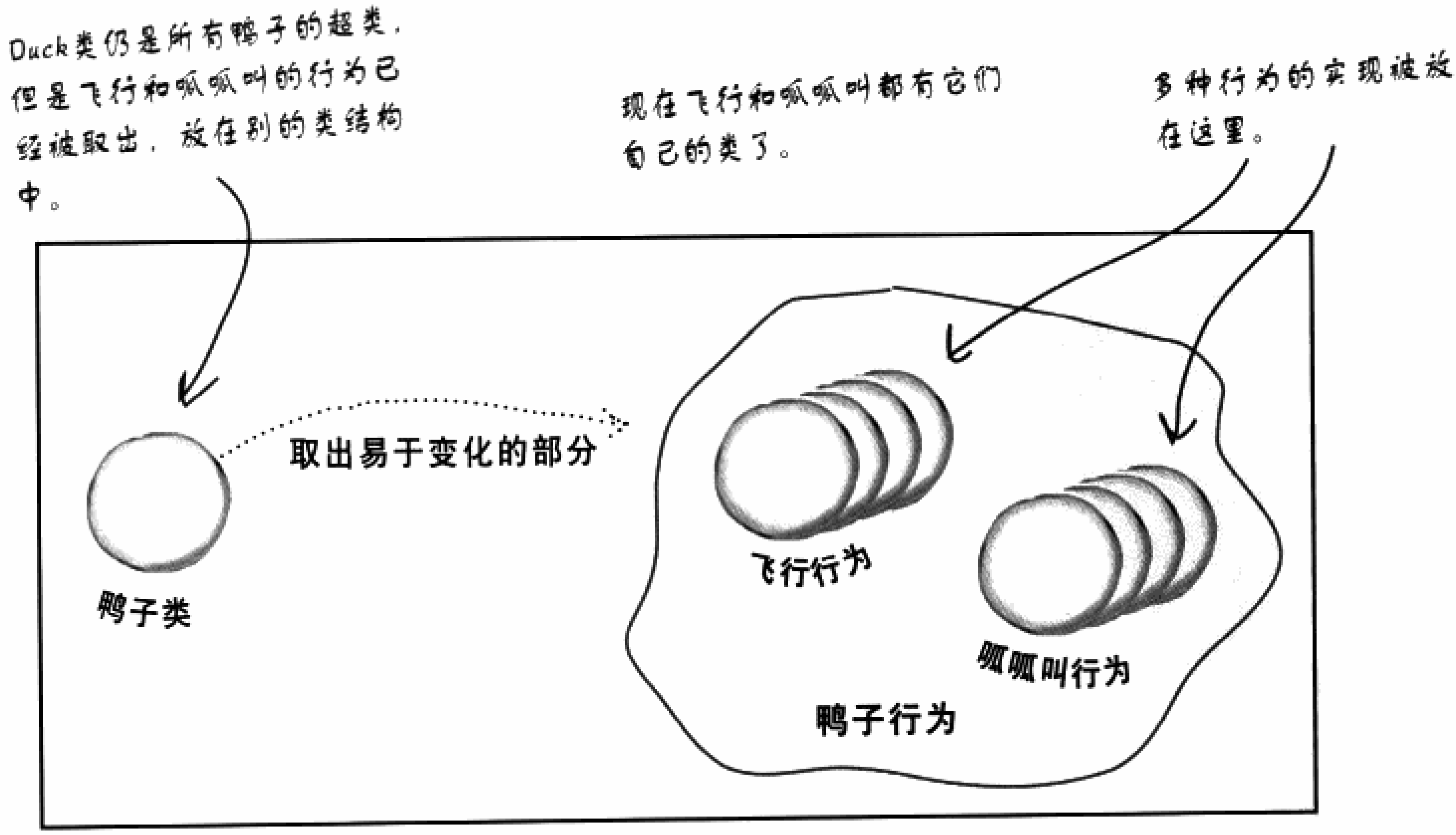
分开变化和不会变化的部分

从哪里开始呢？就我们目前所知，除了fly()和quack()的问题之外，Duck类还算一切正常，似乎没有特别需要经常变化或修改的地方。所以，除了某些小改变之外，我们不打算对Duck类做太多处理。

现在，为了要分开“变化和不会变化的部分”，我们准备建立两组类（完全远离Duck类），一个是“fly”相关的，一个是“quack”相关的，每一组类将实现各自的动作。比方说，我们可能有一个类实现“呱呱叫”，另一个类实现“吱吱叫”，还有一个类实现“安静”。

我们知道Duck类内的fly()和quack()会随着鸭子的不同而改变。

为了要把这两个行为从Duck类中分开，我们将把它们从Duck类中取出来，建立一组新类来代表每个行为。



设计鸭子的行为

如何设计那组实现飞行和呱呱叫的行为的类呢？

我们希望一切能有弹性，毕竟，正是因为一开始鸭子行为没有弹性，才让我们走上现在这条路。我们还想能够“指定”行为到鸭子的实例。比方说，我们想要产生一个新的绿头鸭实例，并指定特定“类型”的飞行行为给它。干脆顺便让鸭子的行为可以动态地改变好了。换句话说，我们应该在鸭子类中包含设定行为的方法，这样就可以在“运行时”动态地“改变”绿头鸭的飞行行为。

有了这些目标要实现，接着看看第二个设计原则：

设计原则

针对接口编程，而不是针对实现编程。

从现在开始，鸭子的行为将被放在分开的类中，此类专门提供某行为接口的实现。

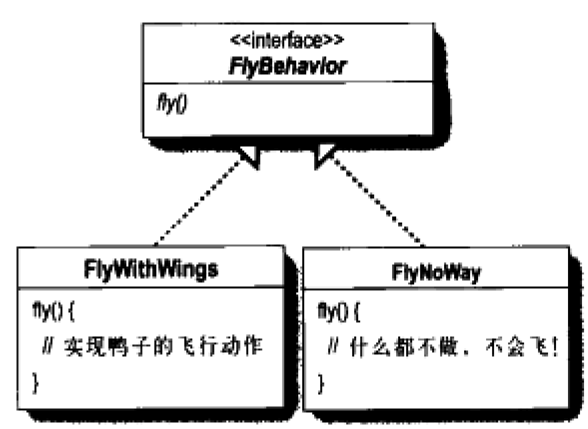
这样，鸭子类就不再需要知道行为的实现细节。

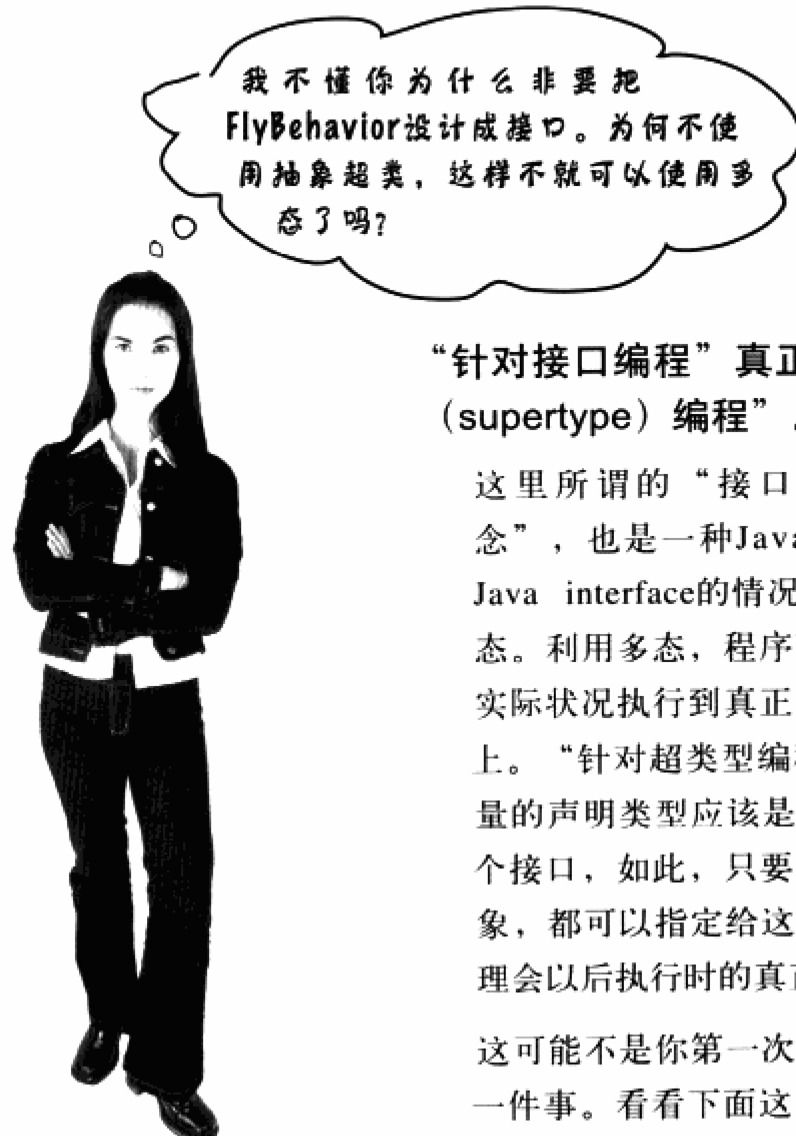
我们利用接口代表每个行为，比方说，FlyBehavior与Quack-Behavior，而行为的每个实现都将实现其中的一个接口。

所以这次鸭子类不会负责实现Flying与Quacking接口，反而是由我们制造一组其他类专门实现FlyBehavior与QuackBehavior，这就称为“行为”类。由行为类而不是Duck类来实现行为接口。

这样的做法迥异于以往，以前的做法是：行为来自Duck超类的具体实现，或是继承某个接口并由子类自行实现而来。这两种做法都是依赖于“实现”，我们被实现绑得死死的，没办法更改行为（除非写更多代码）。

在我们的新设计中，鸭子的子类将使用接口（FlyBehavior与QuackBehavior）所表示的行为，所以实际的“实现”不会被绑死在鸭子的子类中。（换句话说，特定的具体行为编写在实现了FlyBehavior与QuakcBehavior的类中）。

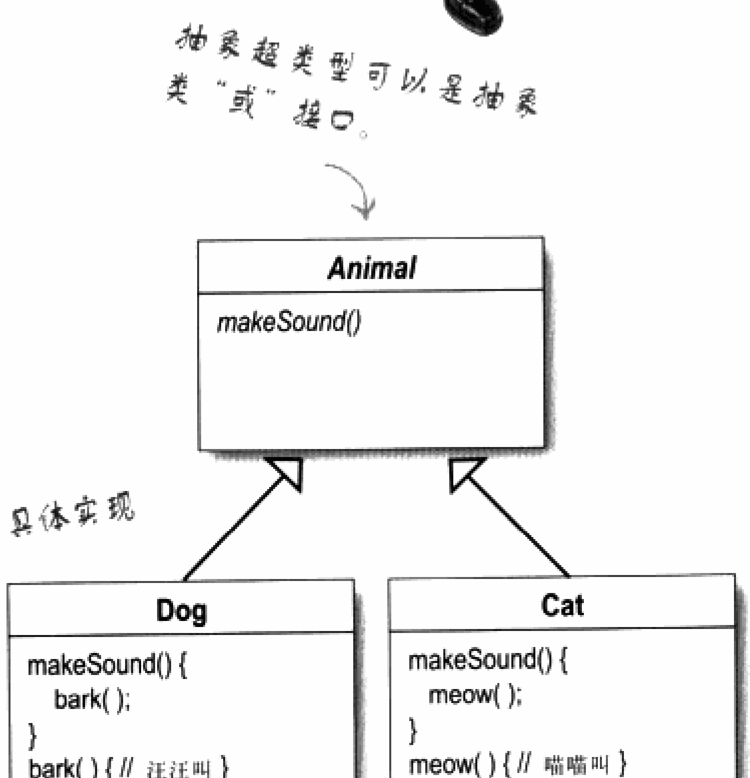




“针对接口编程”真正的意思是“针对超类型（supertype）编程”。

这里所谓的“接口”有多个含义，接口是一个“概念”，也是一种Java的interface构造。你可以在不涉及Java interface的情况下，“针对接口编程”，关键就在多态。利用多态，程序可以针对超类型编程，执行时会根据实际状况执行到真正的行为，不会被绑死在超类型的行为上。“针对超类型编程”这句话，可以更明确地说成“变量的声明类型应该是超类型，通常是一个抽象类或者是一个接口，如此，只要是具体实现此超类型的类所产生的对象，都可以指定给这个变量。这也意味着，声明类时不用理会以后执行时的真正对象类型！”

这可能不是你第一次听到，但是请务必注意我们说的是同一件事。看看下面这个简单的多态例子：假设有一个抽象类Animal，有两个具体的实现（Dog与Cat）继承Animal。做法如下：



“针对实现编程”

```
Dog d = new Dog();
d.bark();
```

声明变量“d”为Dog类型（是Animal的具体实现），会造成我们必须针对具体实现编码。

但是，“针对接口/超类型编程”做法会如下：

```
Animal animal = new Dog();
animal.makeSound();
```

我们知道该对象是狗，但是我们现在利用animal进行多态的调用。

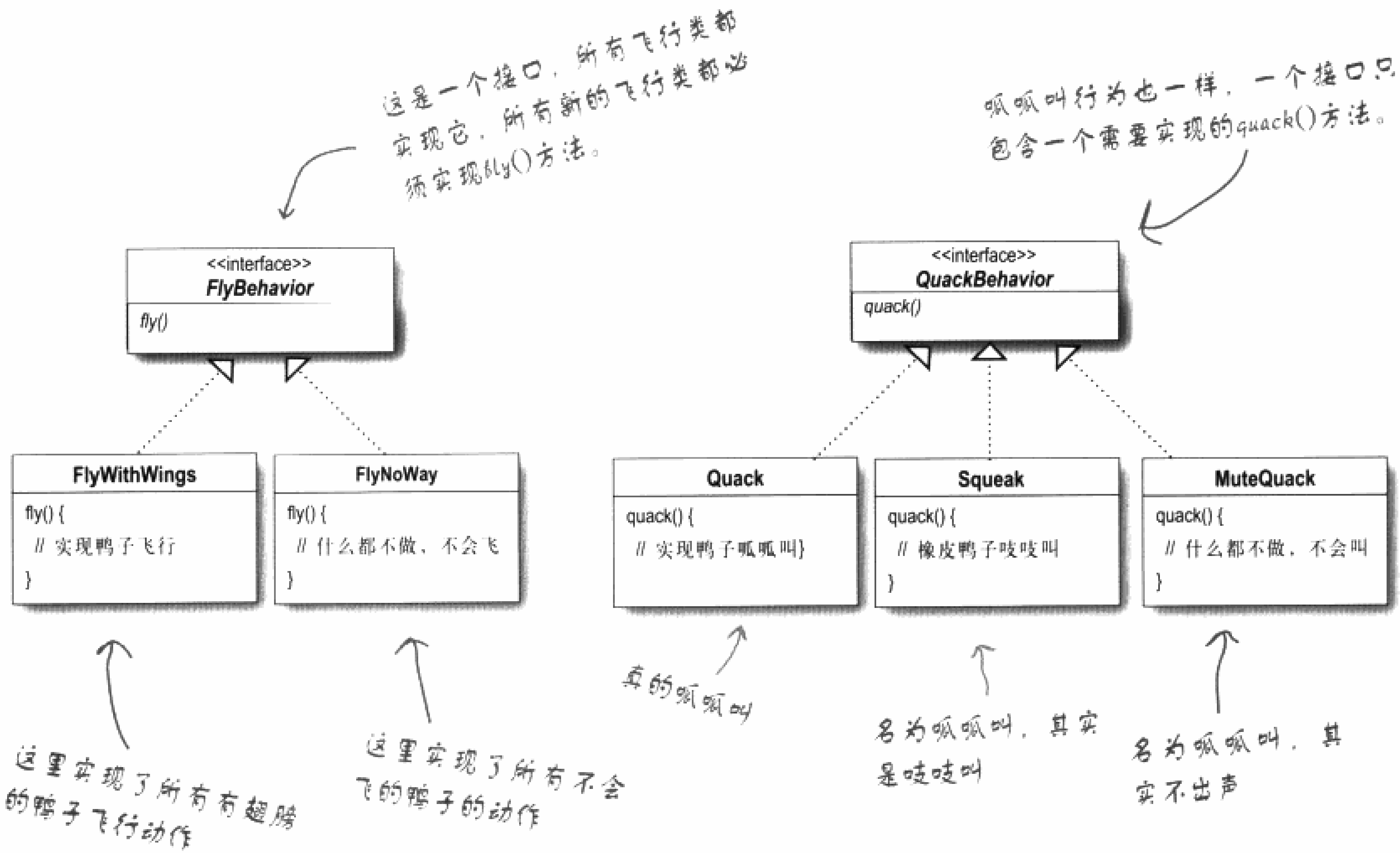
更棒的是，子类实例化的动作不再需要在代码中硬编码，例如new Dog()，而是“在运行时才指定具体实现的对象”。

```
a = getAnimal();
a.makeSound();
```

我们不知道实际的子类型是“什么”……我们只关心它知道如何正确地进行makeSound()的动作就够了。

实现鸭子的行为

在此，我们有两个接口，FlyBehavior和QuackBehavior，还有它们对应的类，负责实现具体的行为：



这样的设计，可以让飞行和呱呱叫的动作被其他的对象复用，因为这些行为已经与鸭子类无关了。

而我们可以新增一些行为，不会影响到既有的行为类，也不会影响“使用”到飞行行为的鸭子类。

这么一来，有了继承的“复用”好处，却没有继承所带来的包袱。

there are no Dumb Questions

问：我是不是一定要先把系统做出来，再看看有哪些地方需要变化，然后才回头去把这些地方分离&封装？

答：不尽然。通常在你设计系统时，预先考虑到有哪些地方未来可能需要变化，于是提前在代码中加入这些弹性。你会发现，原则与模式可以应用在软件开发生命周期的任何阶段。

问：Duck是不是也该设计成一个接口？

答：在本例中，这么做并不恰当。如你所见的，我们已经让一切都整合妥当，而且让Duck成为一个具体类，这样可以让衍生的特定类（例如绿头鸭）具有Duck共同的属性和方法。我们已经从Duck的继承结构中删除了变化的部分，原先的问题都已经解决了，所以不需要把Duck设计成接口。

问：用一个类代表一个行为，感觉似乎有点奇怪。类不是应该代表某种“东西”吗？类不是应该同时具备状态“与”行为吗？

答：在OO系统中，是的，类代表的东西一般都是既有状态（实例变量）又有方法。只是在本例中，碰巧“东西”是个行为。但是即使是行为，也仍然可以有状态和方法，例如，飞行的行为可以具有实例变量，记录飞行行为的属性（每秒翅膀拍动几下、最大高度和速度等）。

Sharpen your pencil

1. 使用我们的新设计，如果你要加上一个火箭动力的飞行动作到SimUDuck 系统中，你该怎么做？

2. 除了鸭子之外，你能够想出有什么类会需要用到呱呱叫的行为？

答案：

1) 建立一个FlyRocketPowered类，实现FlyBehavior接口。

2) 例如：鸭鸣器（DuckCall）（一种会产生鸭叫声的装置）。

整合鸭子的行为

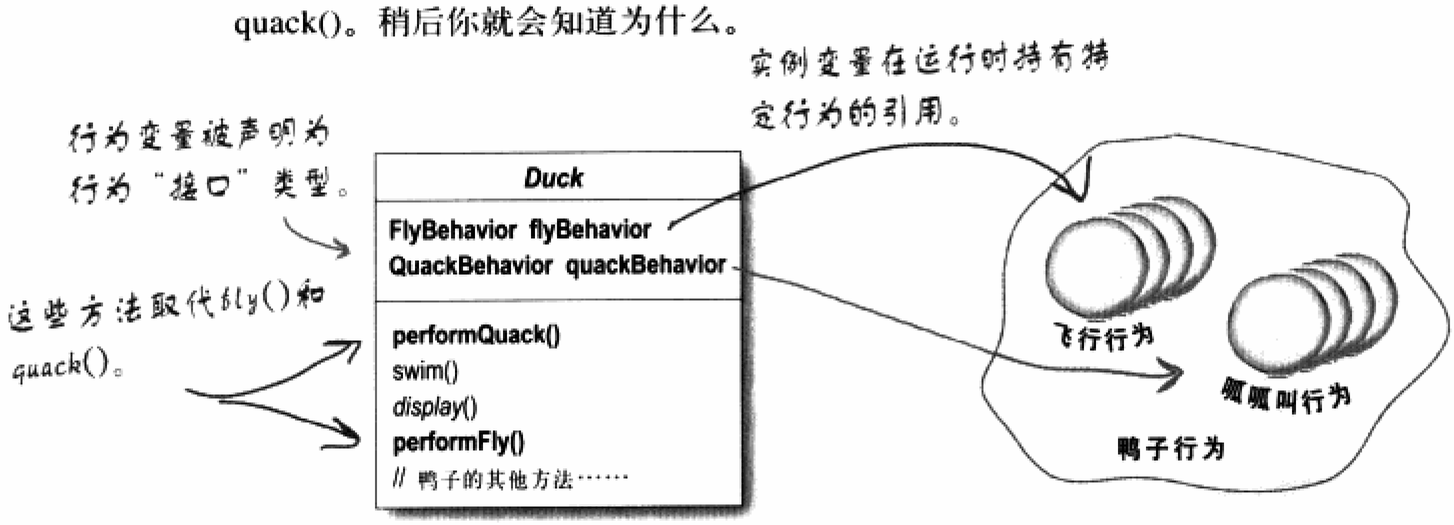
关键在于，鸭子现在会将飞行和呱呱叫的动作“委托”（delegate）别人处理，而不是使用定义在Duck类（或子类）内的呱呱叫和飞行方法。

做法是这样的：

1. **首先，在Duck类中“加入两个实例变量”**，分别为“flyBehavior”与“quack Behavior”，声明为接口类型（而不是具体类实现类型），每个鸭子对象都会动态地设置这些变量以在运行时引用正确的行为类型（例如：FlyWithWings、Squeak等）。

   我们也必须将Duck类与其所有子类中的fly()与quack()删除，因为这些行为已经被搬到FlyBehavior与QuackBehavior类中了。

   我们用两个相似的方法performFly()和performQuack()取代Duck类中的fly()与quack()。稍后你就会知道为什么。



2. **现在，我们来实现performQuack()：**

```
public class Duck {
   QuackBehavior quackBehavior;
   // 还有更多

   public void performQuack() {
     quackBehavior.quack();
   }
}
```

每只鸭子都会引用实现QuackBehavior接口的对象。

鸭子对象不亲自处理呱呱叫行为，而是委托给quackBehavior引用的对象。

很容易，是吧？想进行呱呱叫的动作，Duck对象只要叫quackBehavior对象去呱呱叫就可以了。在这部分的代码中，我们不在乎quackBehavior接口的对象到底是什么，我们只关心该对象知道如何进行呱呱叫就够了。

更多的整合……

❸ 好吧！现在来关心"如何设定flyBehavior与quackBehavior的实例变量"。看看MallardDuck类：

```
public class MallardDuck extends Duck {

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();
   }

   public void display() {
       System.out.println("I'm a real Mallard duck");
   }
}
```

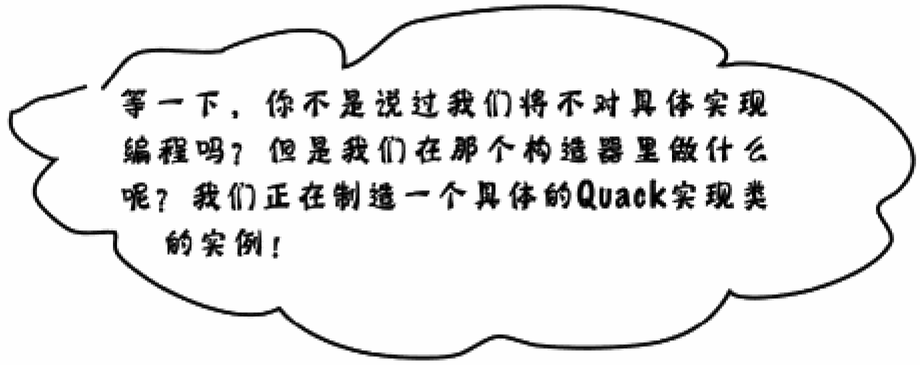
绿头鸭使用Quack类处理呱呱叫，所以当performQuack()被调用时，叫的职责被委托给Quack对象，而我们得到了真正的呱呱叫。

使用FlyWithWings作为其FlyBehavior类型。

别忘了，因为MallardDuck继承Duck类，所以具有flyBehavior与quackBehavior实例变量。

所以，绿头鸭会真的"呱呱叫"，而不是"吱吱叫"，或"叫不出声"。这是怎么办到的？当MallardDuck实例化时，它的构造器会把继承来的quackBehavior实例变量初始化成Quack类型的新实例（Quack是QuackBehavior的具体实现类）。

同样的处理方式也可以用在飞行行为上：MallardDuck的构造器将flyBehavior实例变量初始化成FlyWithWings类型的实例（FlyWithWings是FlyBehavior的具体实现类）。





被你逮到了，我们的确是这么做的……“只是暂时”。

在本书的后续内容中，我们的工具箱中会有更多的模式可用，到时候就可以修正这一点了。

仍请注意，虽然我们把行为设定成具体的类（通过实例化类似Quack或FlyWithWings的行为类，并把它指定到行为引用变量中），但是还是可以在运行时“轻易地”改变它。

所以，目前的做法还是很有弹性的，只是初始化实例变量的做法不够弹性罢了。但是想一想，因为quackBehavior的实例变量是一个接口类型，我们能够在运行时，通过多态的魔力动态地给它地指定不同的QuickBehavior实现类。

花一点儿时间想一想，你如何实现一个其行为可以在运行时改变的鸭子。（几页以后，你就会看到做这件事的代码。）

测试Duck的代码

❶ **输入并编译下面的Duck类（Duck.java）以及两页前的MallardDuck类（MallardDuck.java）。**

```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

为行为接口类型声明两个引用变量，所有鸭子子类（在同一个package中）都继承它们。

委托给行为类

❷ **输入并编译FlyBehavior接口（FlyBehavior.java）与两个行为实现类（FlyWithWings.java与FlyNoWay.java）。**

```
public interface FlyBehavior {
    public void fly();
}
```

所有飞行行为类必须实现的接口。

```
public class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying!!");
    }
}
```

这是飞行行为的实现，给"真会"飞的鸭子用……

```
public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

这是飞行行为的实现，给"不会"飞的鸭子用（包括橡皮鸭和诱饵鸭）。

继续测试Duck的代码 ……

❸ 输入并编译QuackBehavior接口（QuackBehavior.java）及其三个实现类（Quack.java、MuteQuack.java、Squeak.java）。

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Squeak");
   }
}
```

❹ 输入并编译测试类（MiniDuckSimulator.java）

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

这会调用MallardDuck继承来的performQuack()方法，进而委托给该对象的QuackBehavior对象处理（也就是说，调用继承来的quackBehavior引用对象的quack()）。

至于performFly()，也是一样的道理。

❺ 运行代码！

```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

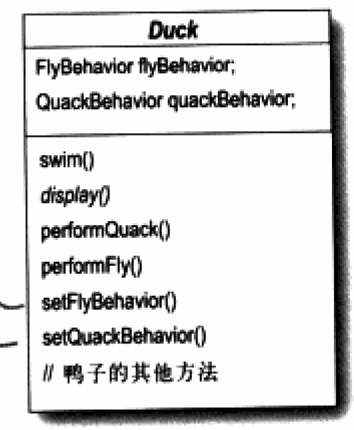
动态设定行为

在鸭子里建立了一堆动态的功能没有用到，就太可惜了！假设我们想在鸭子子类中通过“设定方法（setter method）”来设定鸭子的行为，而不是在鸭子的构造器内实例化。

❶ 在Duck类中，加入两个新方法：

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



从此以后，我们可以“随时”调用这两个方法改变鸭子的行为。

❷ 制造一个新的鸭子类型：模型鸭（ModelDuck.java）

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

一开始，我们的模型鸭是不会飞的。

❸ 建立一个新的FlyBehavior 类型（FlyRocketPowered.java）

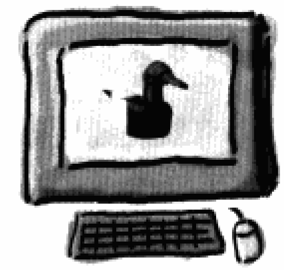
没关系，我们建立一个利用火箭动力的飞行行为。

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket!");
    }
}
```



❹ 改变测试类（MiniDuckSimulator.java），加上模型鸭，并使模型鸭具有火箭动力。

改变前



```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

         Duck model = new ModelDuck();
         model.performFly();
         model.setFlyBehavior(new FlyRocketPowered());
         model.performFly();
   }
}
```

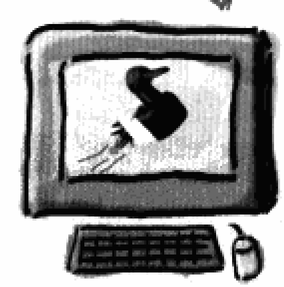
第一次调用performFly()会被委托给flyBehavior对象（也就是FlyNoWay实例），该对象是在模型鸭构造器中设置的。

这会调用继承来的setter方法，把火箭动力飞行的行为设定到模型鸭中。哇！模型鸭突然具有了火箭动力飞行能力！

如果成功了，就意味着模型鸭可以动态地改变它的飞行行为。如果把行为的实现绑死在鸭子类中，可就无法做到这样了。

❺ 运行！

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket!
```



改变后

在运行时想改变鸭子的行为，只需调用鸭子的setter方法就可以。

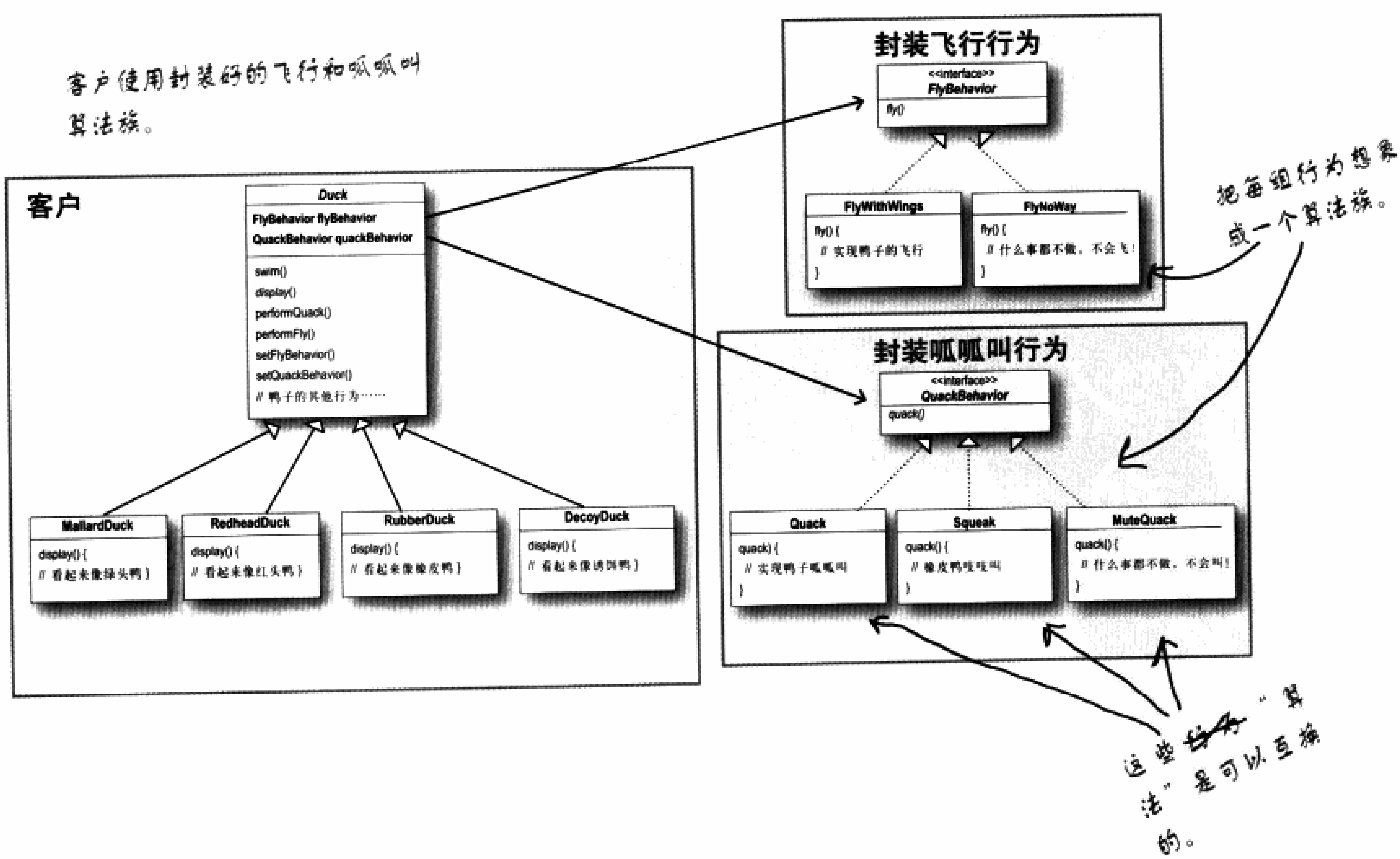
封装行为的大局观

好，我们已经深入研究了鸭子模拟器的设计，该是将头探出水面，呼吸空气的时候了。现在就来看看整体的格局。

下面是整个重新设计后的类结构，你所期望的一切都有：鸭子继承Duck，飞行行为实现FlyBehavior接口，呱呱叫行为实现QuackBehavior接口。

也请注意，我们描述事情的方式也稍有改变。不再把鸭子的行为说成是“一组行为”，我们开始把行为想成是“一族算法”。想想看，在SimUDuck的设计中，算法代表鸭子能做的事（不同的叫法和飞行法），这样的做法也能很容易地用于用一群类计算不同州的销售税金。

请特别注意类之间的“关系”。拿起笔，把下面图形中的每个箭头标上适当的关系，关系可以是IS-A（是一个）、HAS-A（有一个）或IMPLEMENTS（实现）。



“有一个”可能比“是一个”更好。

“有一个”关系相当有趣：每一鸭子都有一个FlyBehavior和一个QuackBehavior，好将飞行和呱呱叫委托给它们代为处理。

当你将两个类结合起来使用，如同本例一般，这就是组合（composition）。这种做法和“继承”不同的地方在于，鸭子的行为不是继承来的，而是和适当的行为对象“组合”来的。

这是一个很重要的技巧。其实是使用了我们的第三个设计原则：

设计原则

多用组合，少用继承。

如你所见，使用组合建立系统具有很大的弹性，不仅可将算法族封装成类，更可以“在运行时动态地改变行为”，只要组合的行为对象符合正确的接口标准即可。

组合用在“许多”设计模式中，在本书中，你也会看到它的诸多优点和缺点。

BRAIN POWER

鸭鸣器（duckcall）是一种装置，猎人用鸭鸣器模拟出鸭叫声，以引诱野鸭。你如何实现你自己的鸭鸣器，而不继承Duck类？



大师与门徒……

大师：蚱蜢，告诉我，在面向对象的道路上，你学到了什么？

门徒：大师，我学到了，面向对象之路承诺了“复用”。

大师：继续说……

门徒：大师，借由继承，好东西可以一再被利用，所以程序开发时间就会大幅减少，就好像在林中很快地砍竹子一样。

大师：蚱蜢呀！软件开发完成“前”以及完成“后”，何者需要花费更多时间呢？

门徒：答案是“后”，大师。我们总是需要花许多时间在系统的维护和变化上，比原先开发花的时间更多。

大师：蚱蜢，这就对啦！那么我们是不是应该致力于提高可维护性和可扩展性上的复用程度呀？

门徒：是的，大师，的确是如此。

大师：我觉得你还有很多东西要学，希望你再深入研究继承。你会发现，继承有它的问题，还有一些其他的方式可以达到复用。

讲到设计模式……



恭喜你，学会第一个模式了！

你刚刚用了你的第一个设计模式：也就是策略模式（Strategy Pattern）。不要怀疑，你正是使用策略模式改写SimUDuck 程序的。多亏这个模式，现在系统不担心遇到任何改变，主管们可以勾画他们的赌城狂欢之旅了。

为了介绍这个模式，我们走了很长的一段路。下面是此模式的正式定义：

策略模式定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

当你需要给朋友留下深刻的印象，或是想影响关键主管的决策时，请使用"这个"定义。

设计谜题

在下面，你将看到一堆杂乱的类与接口，这取自一个动作冒险游戏。你将看到代表游戏角色的类和角色可以使用的武器行为的类。每个角色一次只能使用一种武器，但是可以在游戏的过程中换武器。你的工作是要弄清楚这 ·切……

（答案在本章结尾处）

你的任务：

1. 安排类。
2. 找出一个抽象类、一个接口，以及八个类。
3. 在类之间画箭头。
   a. 继承就画成这样（“extend”）。
   b. 实现接口就画成这样（“implement”）。
   c. “有一个”关系就画成这样。
4. 把setWeapon()方法放到正确的类中。




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

在附近餐厅中无意间听到……



这两人点的餐有何不同？其实没有差异，都是一份单，只是Alice讲话的长度多了一倍，而且快餐店的厨师已经感到不耐烦了。

什么是Flo有的，而Alice没有？答案是，Flo和厨师之间有“共享的词汇”，Alice却不懂这些词汇。共享的词汇不仅方便顾客点餐，也让厨师不用记太多事，毕竟这些餐点模式都已经在他的脑海中了呀！

设计模式让你和其他开发人员之间有共享的词汇，一旦懂得这些词汇，和其他开发人员之间沟通就很容易，也会促使那些不懂的程序员想开始学习设计模式。设计模式也可以把你的思考架构的层次提高到模式层面，而不是仅停留在琐碎的对象上。

在办公室隔间中无意间听到……



BRAIN POWER

除了面向对象设计和在餐厅点餐之外，你还能够想到有哪些例子需要共享词汇？（暗示：想一想汽车修理工、木工、大厨、航管）利用这些行话进行沟通的质量如何？

你能否想到OO设计的什么方面，能够和模式名称匹配的？“策略模式”这个名字是否传神？



共享模式词汇的威力

你使用模式和他人沟通时，其实“不只是”和他人共享“行话”而已。

共享的模式词汇“威力强大”。当你使用模式名称和其他开发人员或者开发团队沟通时，你们之间交流的不只是模式名称，而是一整套模式背后所象征的质量、特性、约束。

“我们使用策略模式实现鸭子的各种行为。”这句话也就是告诉我们，鸭子的行为被封装进入一组类中，可以被轻易地扩充与改变。如果需要，甚至在运行时也可以改变行为。

模式能够让你用更少的词汇做更充分的沟通。当你用模式描述的时候，其他开发人员便很容易地知道你对设计的想法。

想想看，有多少次的设计会议中，你们一不小心就进入了琐碎的实现细节的讨论上。

将说话的方式保持在模式层次，可让你待在“设计圈子”久一点。使用模式谈论软件系统，可以让你保持在设计层次，不会被压低到对象与类这种琐碎的事情上面。

当你的团队开始利用模式分享设计想法与经验，你等于是建立了一个模式使用者的社区。

共享词汇可帮你的开发团队快速充电。对于设计模式有深入了解的团队，彼此之间对于设计的看法不容易产生误解。

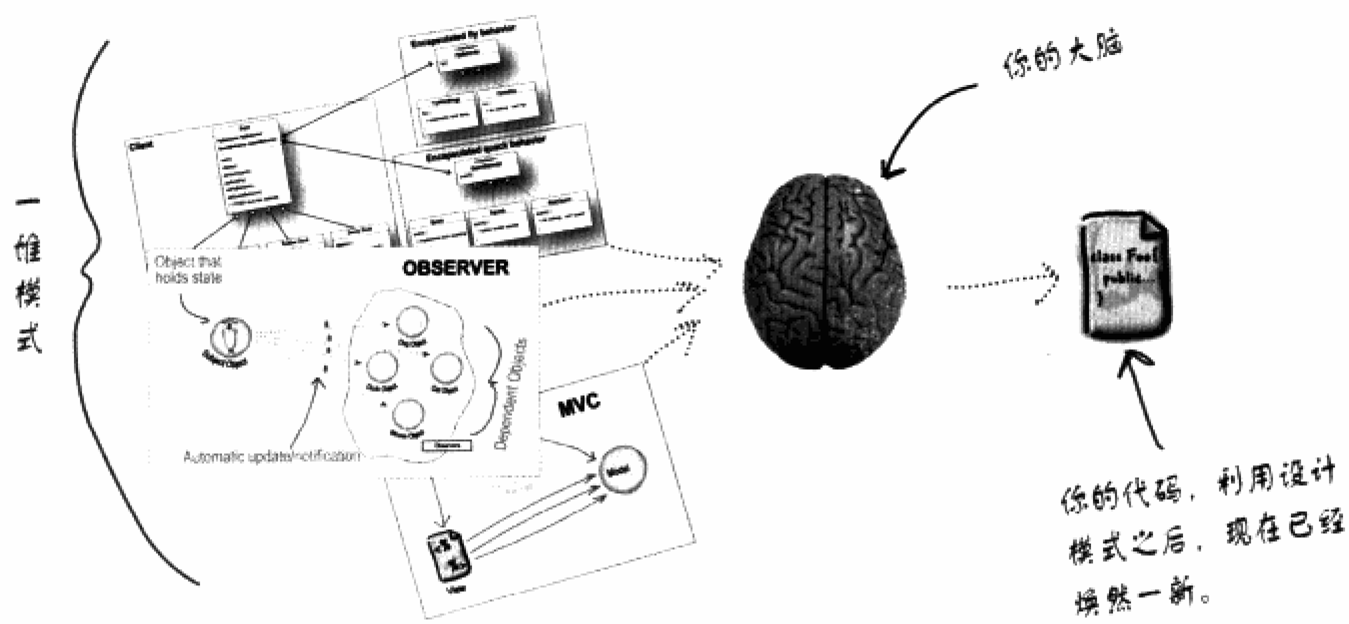
考虑在你的组织内发起一个设计模式研讨会，说不定在学习的过程中，就开始得到回报了……

共享词汇能帮助初级开发人员迅速成长。初级开发人员向有经验的开发人员看齐。当高级开发人员使用设计模式，初级开发人员也会跟着学。把你的组织建立成一个模式使用者的社区。

我如何使用设计模式？

我们全都使用别人设计好的库与框架。我们讨论库与框架、利用它们的API编译成我们的程序、享受运用别人的代码所带来的优点。看看Java API及它所带来的功能：网络、GUI、IO等。库与框架长久以来，一直扮演着软件开发过程的重要角色，我们从中挑选所要的组件，把它们放进合适的地方。但是……库与框架无法帮助我们将应用组织成容易了解、容易维护、具有弹性的架构，所以需要设计模式。

设计模式不会直接进入你的代码中，而是先进入你的“大脑”中。一旦你先在脑海中装入了许多关于模式的知识，就能够开始在新设计中采用它们，并当你的旧代码变得如同搅和成一团没有弹性的意大利面一样时，可用它们重做旧代码。



there are no Dumb Questions

问：如果设计模式这么棒，为何没有人建立相关的库呢？那样我们就不必自己动手了。

答：设计模式比库的等级更高。设计模式告诉我们如何组织类和对象以解决某种问题。而且采纳这些设计并使它们适合我们特定的应用，是我们责无旁贷的事。

问：库和框架不也是设计模式吗？

答：库和框架提供了我们某些特定的实现，让我们的代码可以轻易地引用，但是这并不算是设计模式。有些时候，库和框架本身会用到设计模式，这样很好，因为一旦你了解了设计模式，会更容易了解这些API是围绕着设计模式构造的。

问：那么，没有所谓设计模式的库？

答：没错，但是稍后你会看到设计模式类目。你可以在应用中利用这些设计模式。





怀疑的开发人员

友善的模式大师

开发人员：好吧！但是不都只是好的面向对象设计吗？我是说，我懂得运用封装、抽象、继承、多态，我真的还有必要考虑设计模式吗？运用OO，一切不是都很直接吗？这不正是我过去上了一堆OO课程的原因吗？我认为设计模式只对那些不懂好的OO设计的人有用。

大师：这是面向对象开发常有的谬误：以为知道OO基础概念，就能自动设计出弹性的、可复用的、可维护的系统。

开发人员：不是这样吗？

大师：不是！要构造有这些特征的OO系统，事实证明只有通过不断地艰苦实践，才能成功。

开发人员：我想我开始了解了，这些构造OO系统的隐含经验于是被收集整理出来……

大师：……是的，被整理成了一群“设计模式”。

开发人员：那么，如果知道了这些模式，我就可以减少许多体力劳动，直接采用可行的模式吗？

大师：对，在一定程度上可以这么说。不过要记住，设计是一门艺术，总是有许多可取舍的地方。但是如果你能采用这些经过深思熟虑，且经受过时间考验的设计模式，你就领先别人了。

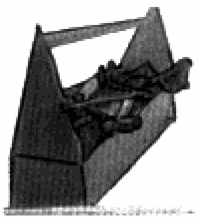


开发人员：如果我找不到模式，怎么办？

大师：有一些面向对象原则，适用于所有的模式。当你无法找到适当的模式解决问题时，采用这些原则可以帮助你。

开发人员：原则？你是说除了抽象、封装……之外，还有其他的？

大师：是的，建立可维护的OO系统，要诀就在于随时想到系统以后可能需要的变化以及应付变化的原则。



设计工具箱内的工具

你几乎快要读完第1章了！你已经在你的设计工具箱内放进了几样工具，在我们进入第2章之前，先将这些工具一一列出。

OO基础

抽象

封装

多态

继承

我们假设你知道OO基础包括了多态的用法、继承就像按契约进行设计、封装是如何运作的。如果你觉得脑袋有一点生锈了，快快拿出你的《Head First Java》复习，然后再把这一章读一遍。

OO原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

我们会在后续的内容中更详细地看看这些原则，还会再多加一些原则到清单上。

OO模式

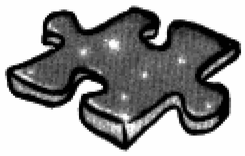
策略模式——定义算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

阅读本书时，时时刻刻要思考着：模式如何仰赖基础与原则。

学了一个，还有更多！

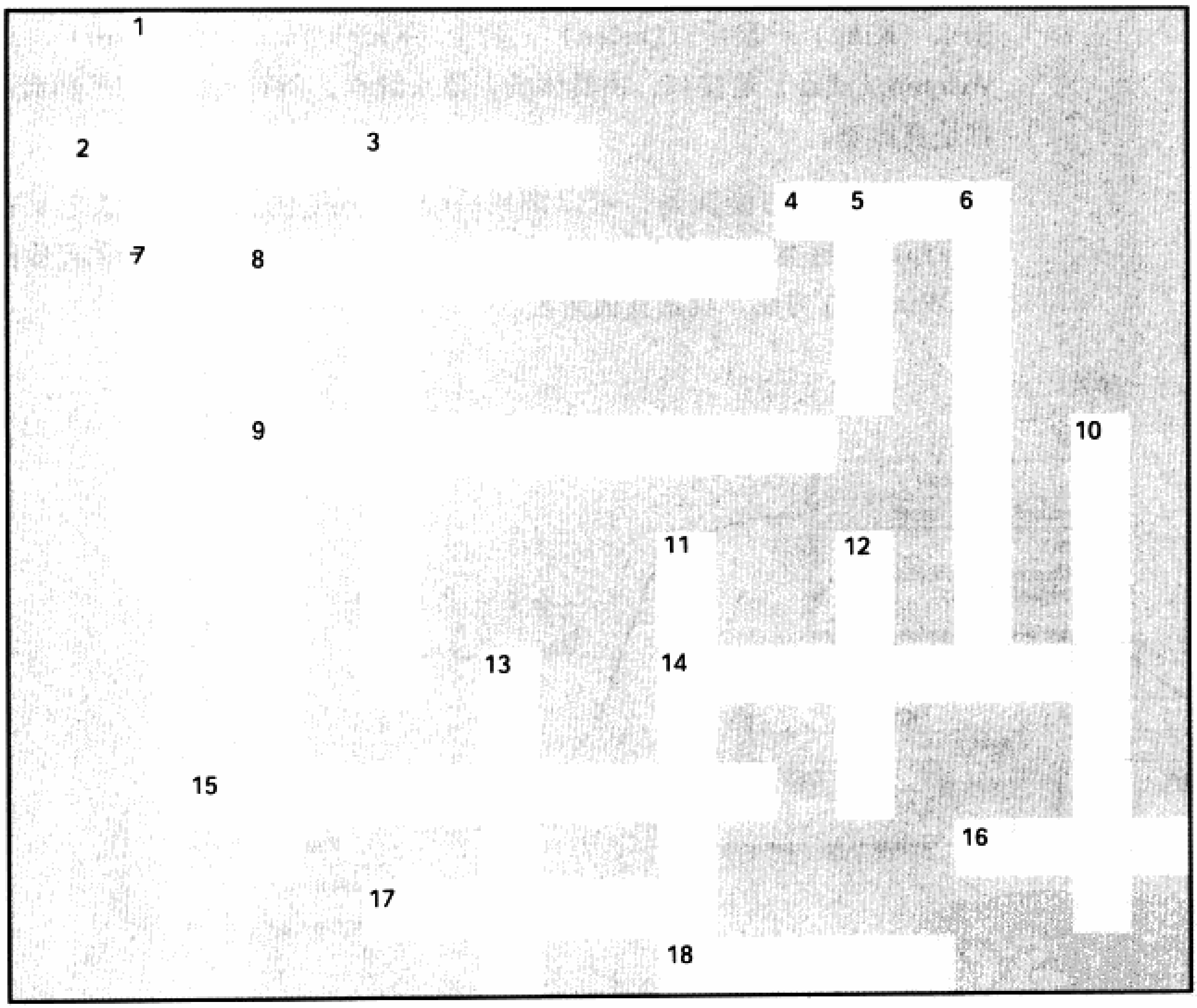
要点

- 知道OO基础，并不足以让你设计出良好的OO系统。
- 良好的OO设计必须具备可复用、可扩充、可维护三个特性。
- 模式可以让我们建造出具有良好OO设计质量的系统。
- 模式被认为是历经验证的OO设计经验。
- 模式不是代码，而是针对设计问题的通用解决方案。你可把它们应用到特定的应用中。
- 模式不是被发明，而是被发现。
- 大多数的模式和原则，都着眼于软件变化的主题。
- 大多数的模式都允许系统局部改变独立于其他部分。
- 我们常把系统中会变化的部分抽出来封装。
- 模式让开发人员之间有共享的语言，能够最大化沟通的价值。



让标准填字游戏，动动你的右脑。

这是一个标准的纵横填字游戏，所有的词都来自本章。



横排提示：

2. Grilled cheese with bacon
4. Duck demo was located where
7. _______ what varies
9. Most patterns follow from OO _______
14. Pattern that fixed the simulator
15. Patterns give us a shared _______
16. Design patterns _____
17. Development constant
18. Patterns ____ in many applications

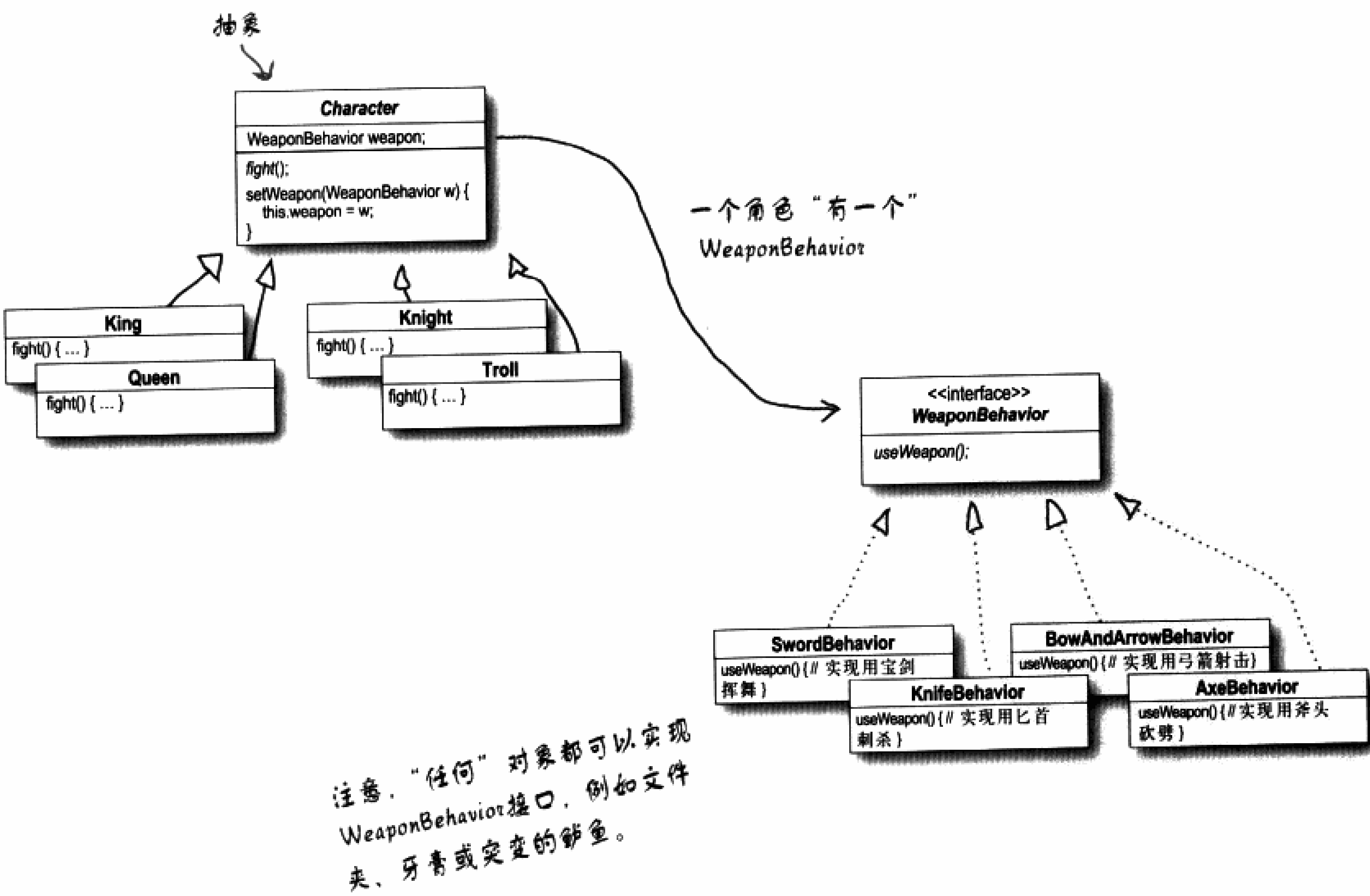
竖排提示：

1. High level libraries
3. Learn from the other guy's _______
5. Java IO, Networking, Sound
6. Program to this, not an implementation
8. Favor over inheritance
10. Duck that can't quack
11. Rick was thrilled with this pattern
12. Patterns go into your _______
13. Rubberducks make a ________

设计谜题解答

Character（角色）是抽象类，由具体的角色来继承。具体的角色包括：国王（King）、皇后（Queen）、骑士（Knight）、妖怪（Troll）。而Weapon（武器）是接口，由具体的武器来继承。所有实际的角色和武器都是具体类。

任何角色如果想换武器，可以调用setWeapon()方法，此方法定义在Character超类中。在打斗（flight）过程中，会调用到目前武器的useWeapon() 方法，攻击其他角色。

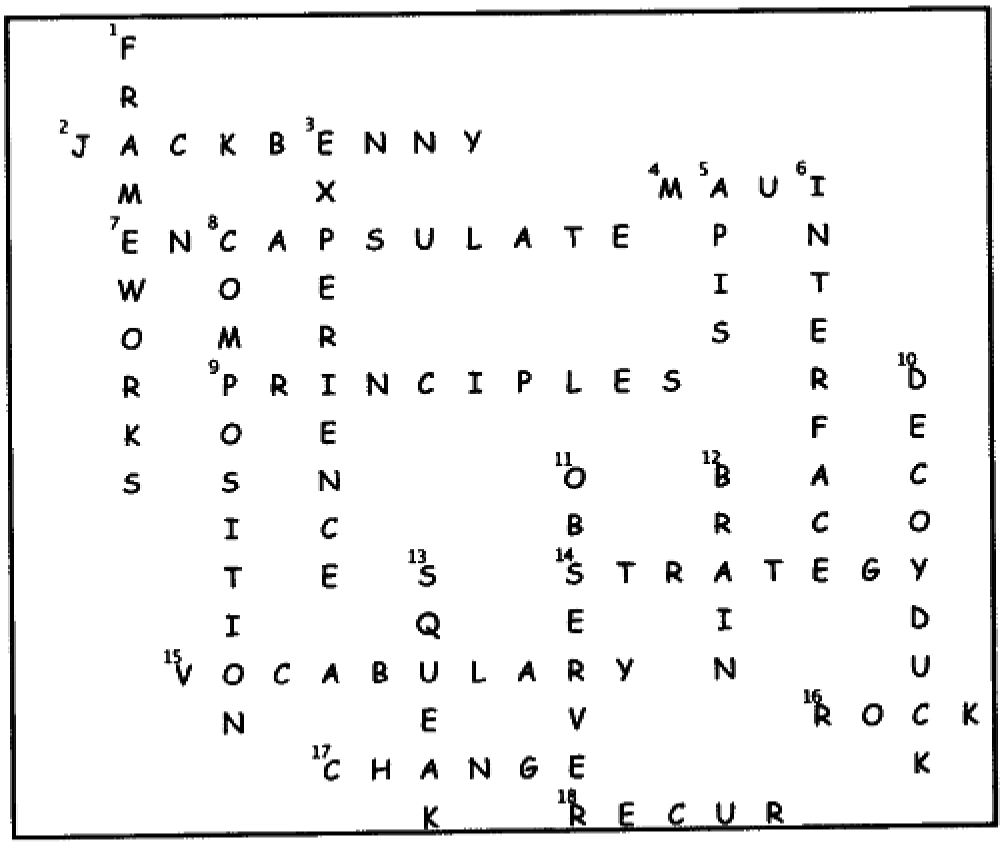


答案

Sharpen your pencil

利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

- ☑ A. 代码在多个子类中重复。
- ☑ B. 运行时的行为不容易改变。
- ☐ C. 我们不能让鸭子跳舞。
- ☑ D. 很难知道所有鸭子的全部行为。
- ☐ E. 鸭子不能同时又飞又叫。
- ☑ F. 改变会牵一发动全身，造成其他鸭子不想要的改变。



Sharpen your pencil

驱动改变的因素很多。找出你的软件中需要改变代码的地方，一一列出来。下面是我们的答案，你的答案可能和我们不一样。

我们的顾客或用户决定要别的做法，或者想要新功能。
我的公司决定采用别的数据库产品，又从另一家厂商买了数据，这造成数据格式不兼容。唉！
唔！应对技术改变，我们必须更新代码，适用于新协议。
我们学到了足够的构建系统的知识，希望回去把事情做得更好。

2 观察者（Observer）模式

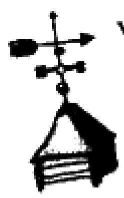
让你的对象知悉现况



有趣的事情发生时，可千万别错过了！ 有一个模式可以帮你的对象知悉现况，不会错过该对象感兴趣的事。对象甚至在运行时可决定是否要继续被通知。观察者模式是JDK中使用最多的模式之一，非常有用。我们也会一并介绍一对多关系，以及松耦合（对，没错，我们说耦合）。有了观察者，你将会消息灵通。

恭喜你！

你的团队刚刚赢得一纸合约，负责建立Weather-O-Rama公司的下一代气象站——Internet气象观测站。



Weather-O-Rama气象站
100 Main Street
Tornado Alley, OK 45021

工作合约

恭喜贵公司获选为敝公司建立下一代Internet气象观测站！

该气象站必须建立在我们专利申请中的WeatherData对象上，由WeatherData对象负责追踪目前的天气状况（温度、湿度、气压）。我们希望贵公司能建立一个应用，有三种布告板，分别显示目前的状况、气象统计及简单的预报。当WeatherObject对象获得最新的测量数据时，三种布告板必须实时更新。

而且，这是一个可以扩展的气象站，Weather-O-Rama气象站希望公布一组API，好让其他开发人员可以写出自己的气象布告板，并插入此应用中。我们希望贵公司能提供这样的API。

Weather-O-Rama气象站有很好的商业营运模式：一旦客户上钩，他们使用每个布告板都要付钱。最好的部分就是，为了感谢贵公司建立此系统，我们将以公司的认股权支付你。

我们期待看到你的设计和应用的alpha版本。

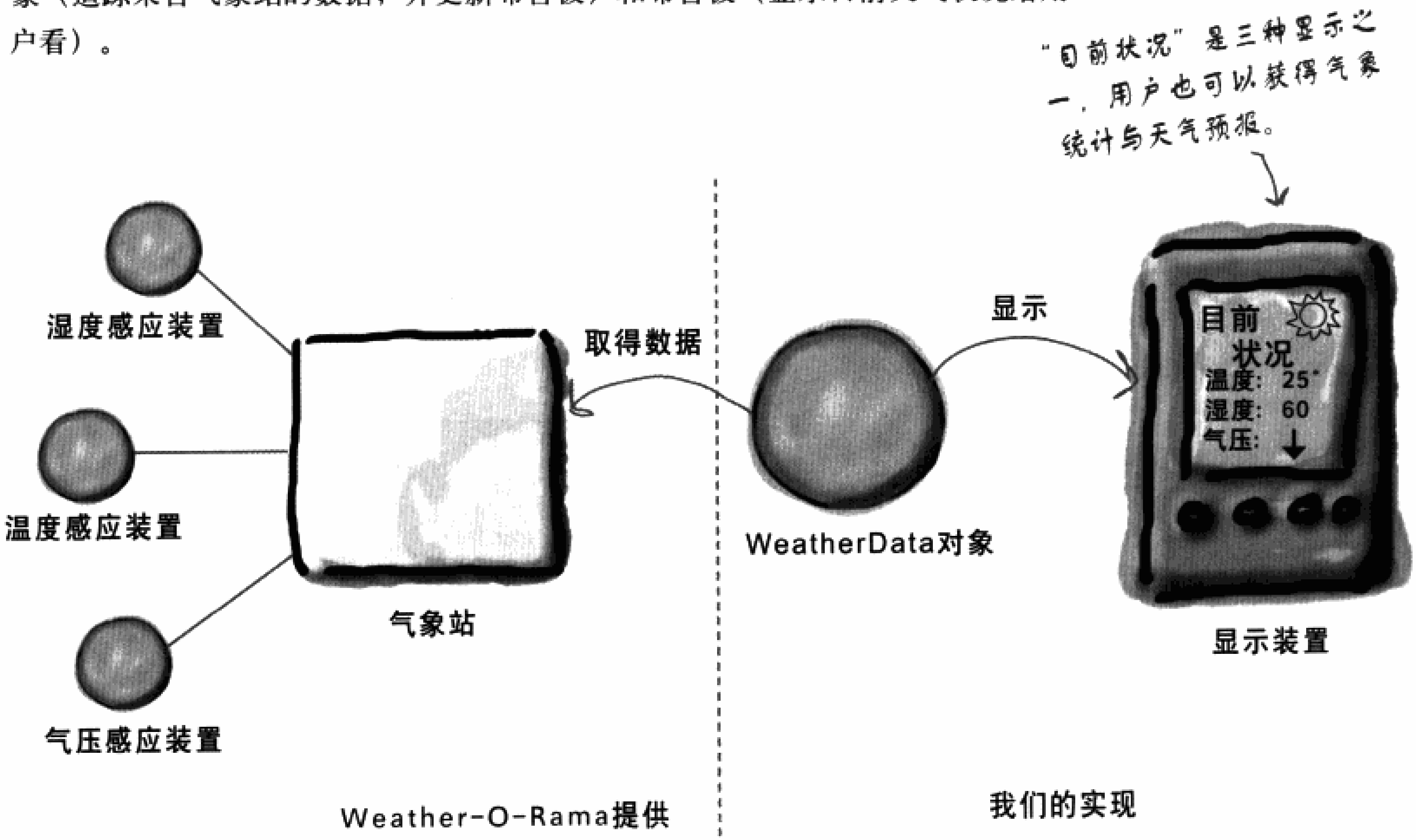
真挚的

Johnny Hurricane——Weather-O-Rama气象站执行长

附注：我们正通宵整理WeatherData源文件给你们。

气象监测应用的概况

此系统中的三个部分是气象站（获取实际气象数据的物理装置）、WeatherData对象（追踪来自气象站的数据，并更新布告板）和布告板（显示目前天气状况给用户看）。



WeatherData对象知道如何跟物理气象站联系，以取得更新的数据。WeatherData对象会随即更新三个布告板的显示：目前状况（温度、湿度、气压）、气象统计和天气预报。

如果我们选择接受这个项目，我们的工作就是建立一个应用，利用WeatherData对象取得数据，并更新三个布告板：目前状况、气象统计和天气预报。

瞧一瞧刚送到的WeatherData类

如同他们所承诺的，隔天早上收到了WeatherData源文件，看了一下代码，一切都很直接：

WeatherData
getTemperature( ) getHumidity( ) getPressure( ) measure- mentsChanged( ) // 其他的方法

这三个方法各自返回最近的气象测量数据（分别为，温度、湿度、气压）。我们不在乎这些变量"如何"被设置，WeatherData对象自己知道如何从气象站获取更新信息。

```
/*
 * 一旦气象测量更新，此方法会被调用
 */
public void measurementsChanged() {
    // 你的代码加在这里
}
```

WeatherData.java

WeatherObject的开发人员留了一个线索，好让我们知道该加些什么……

再次提醒，这只是三个显示布告板中的一个。



显示装置

我们的工作是实现measurementsChanged()，好让它更新目前状况、气象统计、天气预报的显示布告板。

我们目前知道些什么？



Weather-O-Rama气象站的要求说明并不是很清楚，我们必须搞懂该做些什么。那么，我们目前知道些什么呢？

- WeatherData类具有getter方法，可以取得三个测量值：温度、湿度与气压。

```
getTemperature()
getHumidity()
getPressure()
```

- 当新的测量数据备妥时，measurementsChanged()方法就会被调用（我们不在乎此方法是如何被调用的，我们只在乎它被调用了）。

```
measurementsChanged()
```

- 我们需要实现三个使用天气数据的布告板："目前状况"布告、"气象统计"布告、"天气预报"布告。一旦WeatherData有新的测量，这些布告必须马上更新。



第一号布告板



第二号布告板



第三号布告板

- 此系统必须可扩展，让其他开发人员建立定制的布告板，用户可以随心所欲地添加或删除任何布告板。目前初始的布告板有三类："目前状况"布告、"气象统计"布告、"天气预报"布告。



将来的布告板

先看一个错误示范

这是第一个可能的实现：我们依照Weather-O-Rama气象站开发人员的暗示，在measurementsChanged()方法中添加我们的代码：

```
public class WeatherData {

    // 实例变量声明

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // 这里是其他WeatherData方法
}
```

调用WeatherData的三个getXxx()方法，以取得最近的测量值。这些getXxx()方法已经实现好了。

现在，更新布告板……

调用每个布告板更新显示，传入最新的测量。

Sharpen your pencil

在我们的第一个实现中，下列哪种说法正确？（多选）

- ❑ A. 我们是针对具体实现编程，而非针对接口。
- ❑ B. 对于每个新的布告板，我们都得修改代码。
- ❑ C. 我们无法在运行时动态地增加（或删除）布告板。
- ❑ D. 布告板没有实现一个共同的接口。
- ❑ E. 我们尚未封装改变的部分。
- ❑ F. 我们侵犯了WeatherData类的封装。

SWAG的定义:Scientific Wild A** Guess

我们的实现有什么不对？

回想第1章的概念和原则……

```
public void measurementsChanged( ) {

    float temp = getTemperature( );
    float humidity = getHumidity( );
    float pressure = getPressure( );

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

改变的地方，需要封装起来。

至少，这里看起来像是一个统一的接口，布告板的方法名称都是update()，参数都是温度、湿度、气压。

针对具体实现编程，会导致我们以后在增加或删除布告板时必须修改程序。



我们现在就来看观察者模式，然后再回来看看如何将此模式应用到气象观测站。

认识观察者模式

我们看看报纸和杂志的订阅是怎么回事：

1. 报社的业务就是出版报纸。

2. 向某家报社订阅报纸，只要他们有新报纸出版，就会给你送来。只要你是他们的订户，你就会一直收到新报纸。

3. 当你不想再看报纸的时候，取消订阅，他们就不会再送新报纸来。

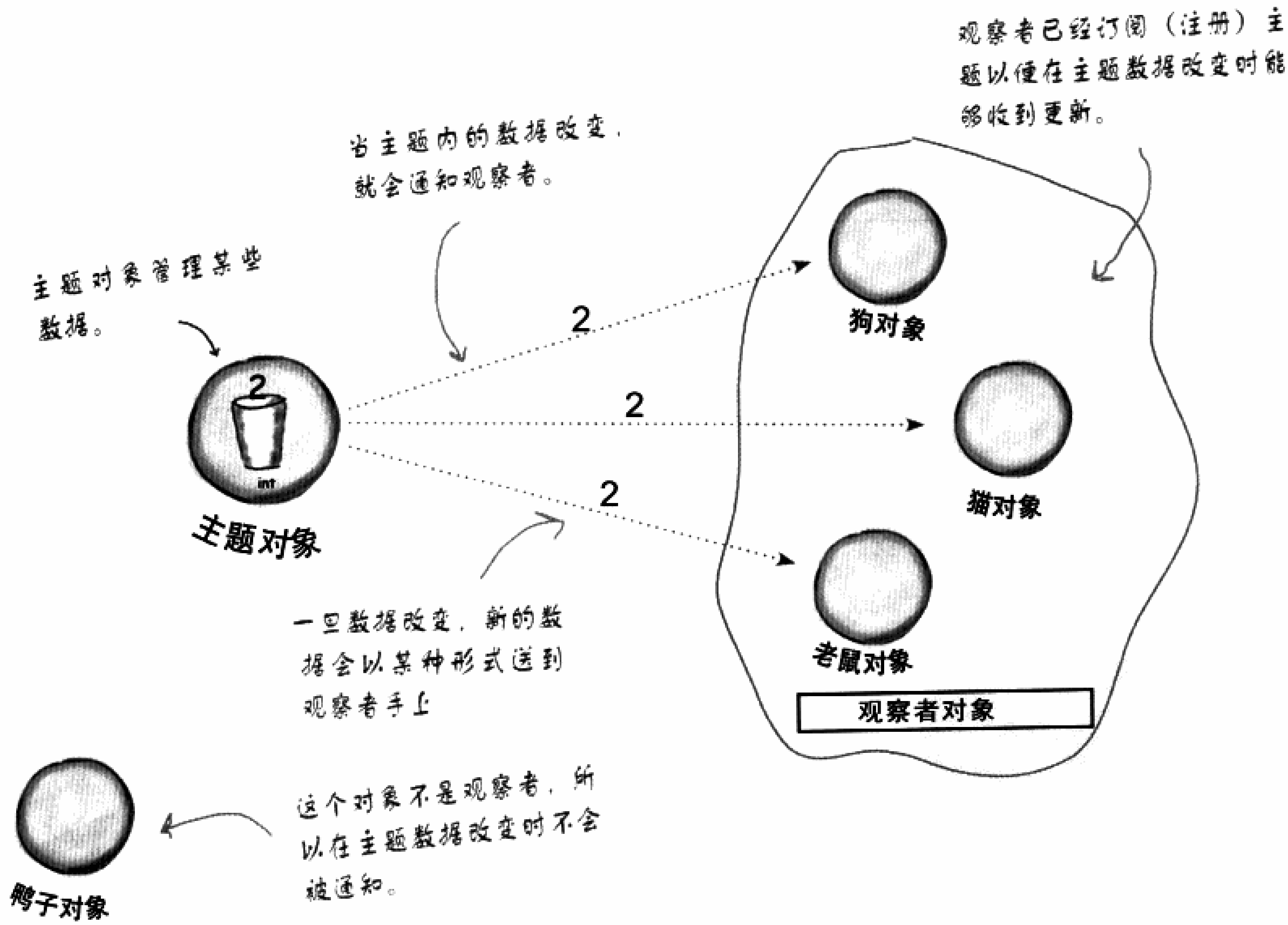
4. 只要报社还在运营，就会一直有人（或单位）向他们订阅报纸或取消订阅报纸。



出版者+订阅者=观察者模式

如果你了解报纸的订阅是怎么回事，其实就知道观察者模式是怎么回事，只是名称不太一样：出版者改称为“主题”（Subject），订阅者改称为“观察者”（Observer）。

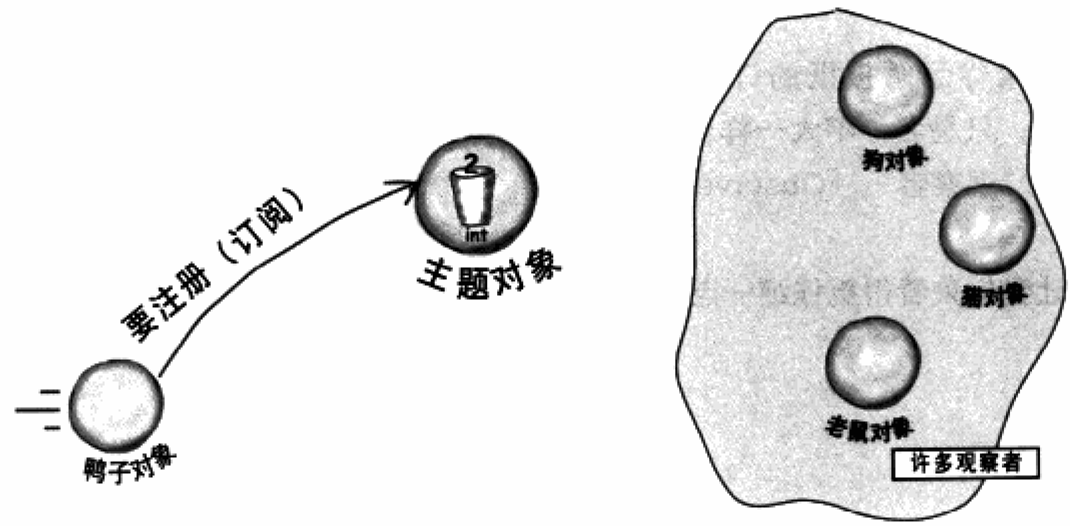
让我们来看得更仔细一点：



观察者模式的一天

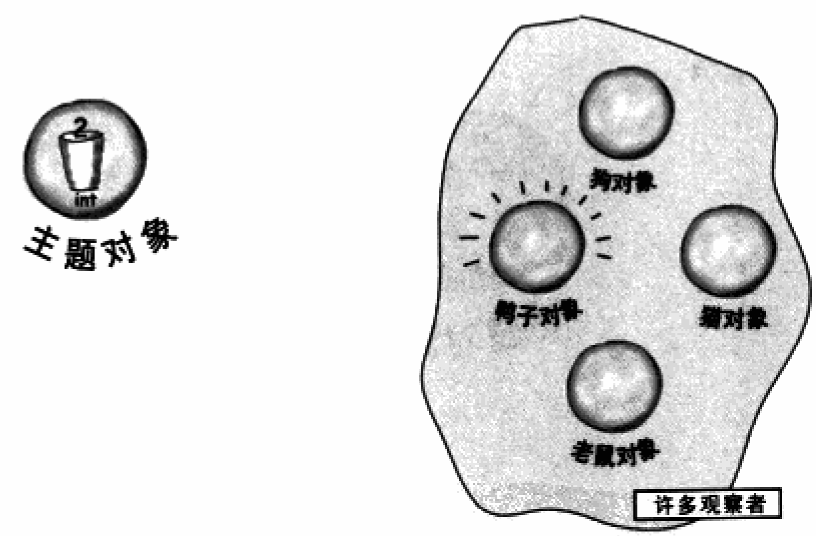
鸭子对象过来告诉主题，它想当一个观察者。

鸭子其实想说的是：我对你的数据改变感兴趣，一有变化请通知我。



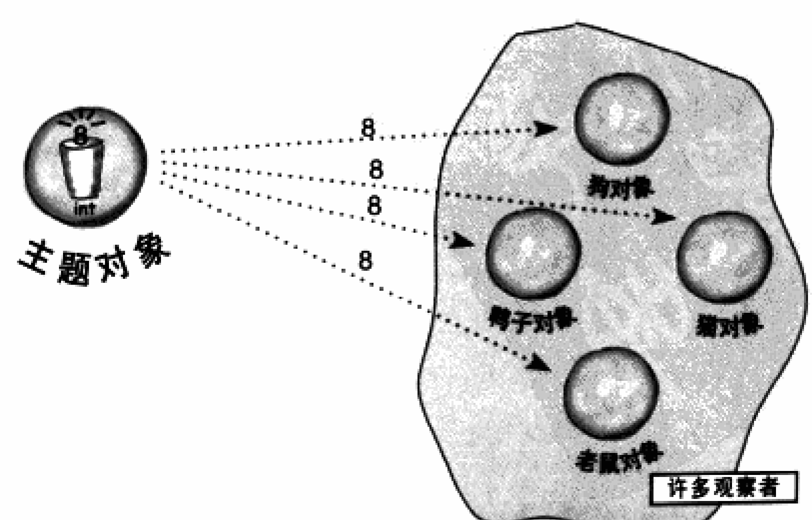
鸭子对象现在已经是正式的观察者了。

鸭子静候通知，等待参与这项伟大的事情。一旦接获通知，就会得到一个整数。



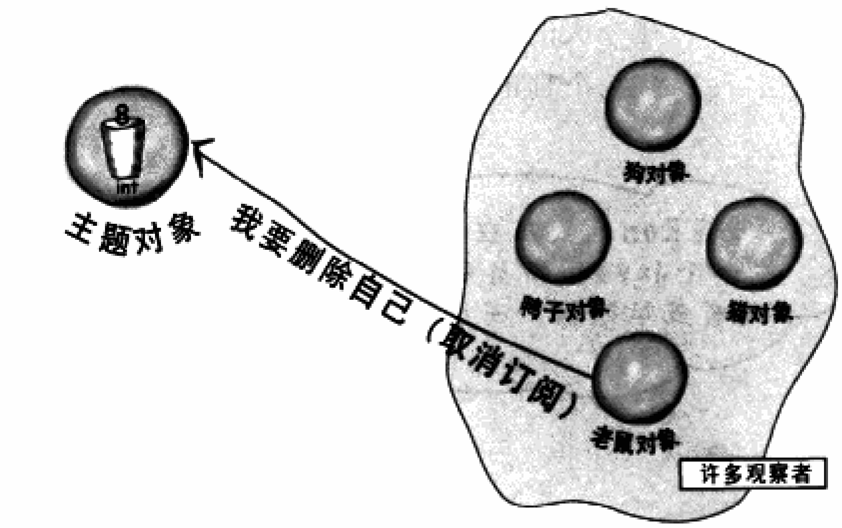
主题有了新的数据值！

现在鸭子和其他所有观察者都会收到通知：主题已经改变了。



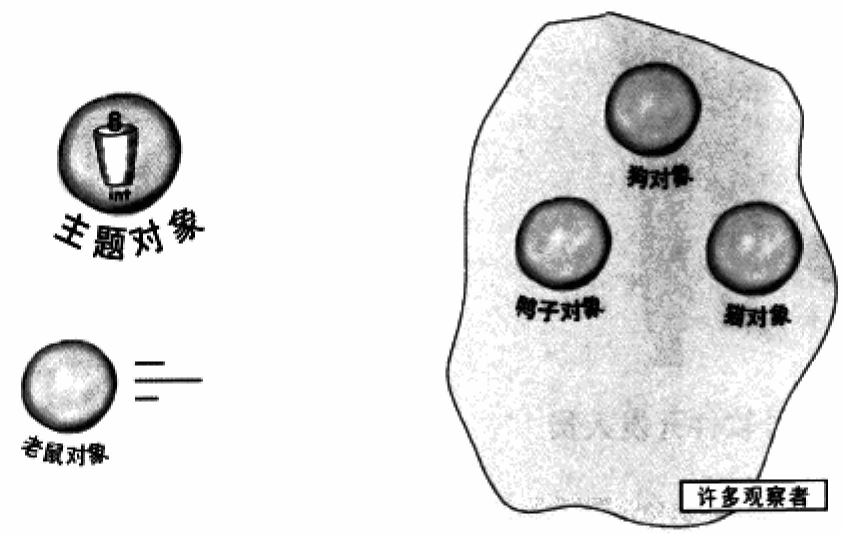
老鼠对象要求从观察者中把自己除名。

老鼠已经观察此主题太久，厌倦了，所以决定不再当个观察者。



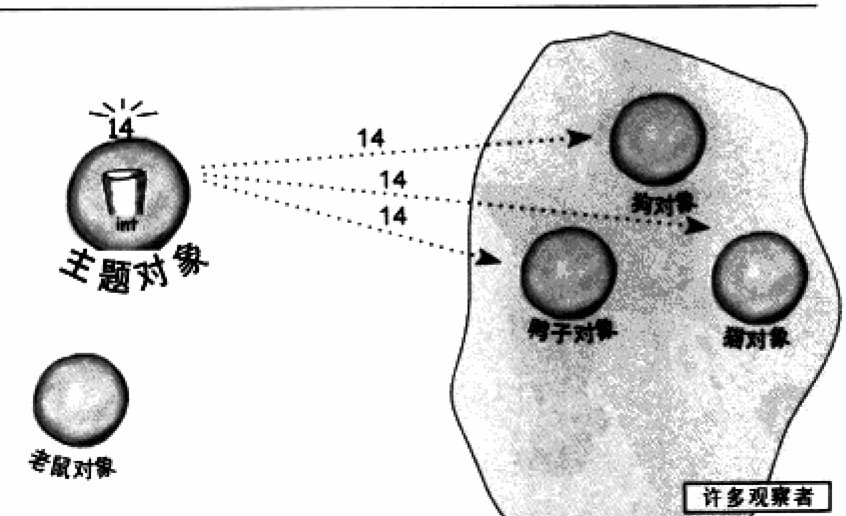
老鼠离开了！

主题知道老鼠的请求之后，把它从观察者中除名。



主题有一个新的整数。

除了老鼠之外，每个观察者都会收到通知，因为它已经被除名了。嘘！不要告诉别人，老鼠其实心中暗暗地怀念这些整数，或许哪天又会再次注册，回来继续当观察者呢！



五分钟短剧：观察的主题

在今天的讽刺短剧中，有两个后泡沫时期的软件工程师，遇到一个真正的猎头……



一号软件开发人员



猎头/主题



二号软件开发人员



主题

5 其间，Ron和Jill继续过自己的日子，如果Java工作来了，他们会接到通知，毕竟，他们是观察者嘛！



6

主题



7

观察者

观察者

Jill自己找到工作了！



8

观察者



9

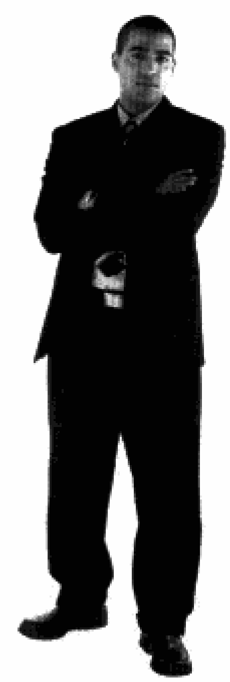
主题

两周后……



Jill热爱她的现状，她不再是观察者了。她还因为签约获得了一笔奖金，因为公司不用拨出一大笔钱给猎头。

但是，我们亲爱的Ron，又如何了？我们听说他设局把原来的猎头搞得毫无招架之力。他不只是一个观察者，也有了自己的求职者清单，只要付一笔钱给猎头，就可从其他求职者赚取更多钱。Ron既是一个主题，也是一个观察者，集两种角色于一身。



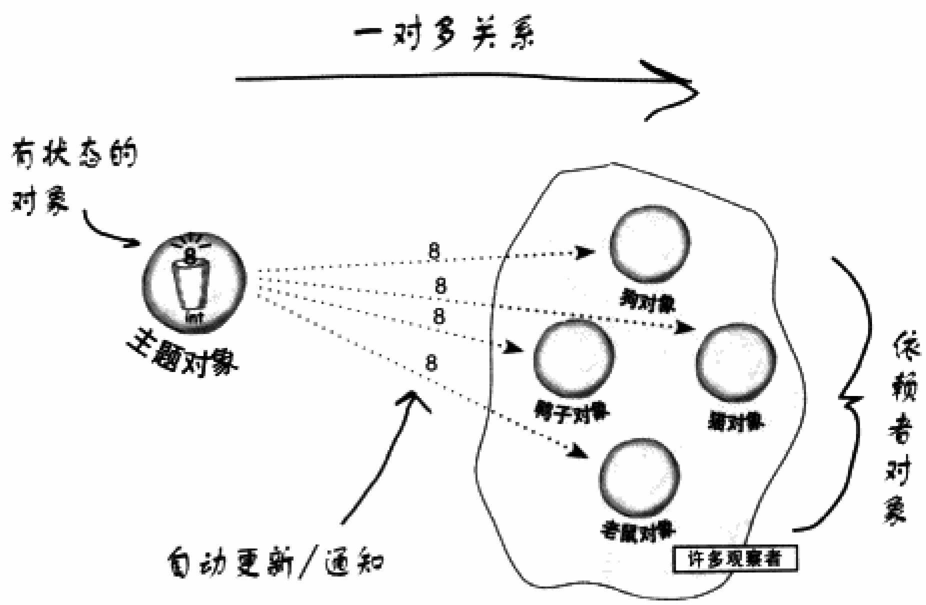
定义观察者模式

当你试图勾勒观察者模式时，可以利用报纸订阅服务，以及出版者和订阅者比拟这一切。

在真实的世界中，你通常会看到观察者模式被定义成：

> **观察者模式**定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

让我们看看这个定义，并和之前的例子做个对照：



主题和观察者定义了一对多的关系。观察者依赖于此主题，只要主题状态一有变化，观察者就会被通知。根据通知的风格，观察者可能因此新值而更新。

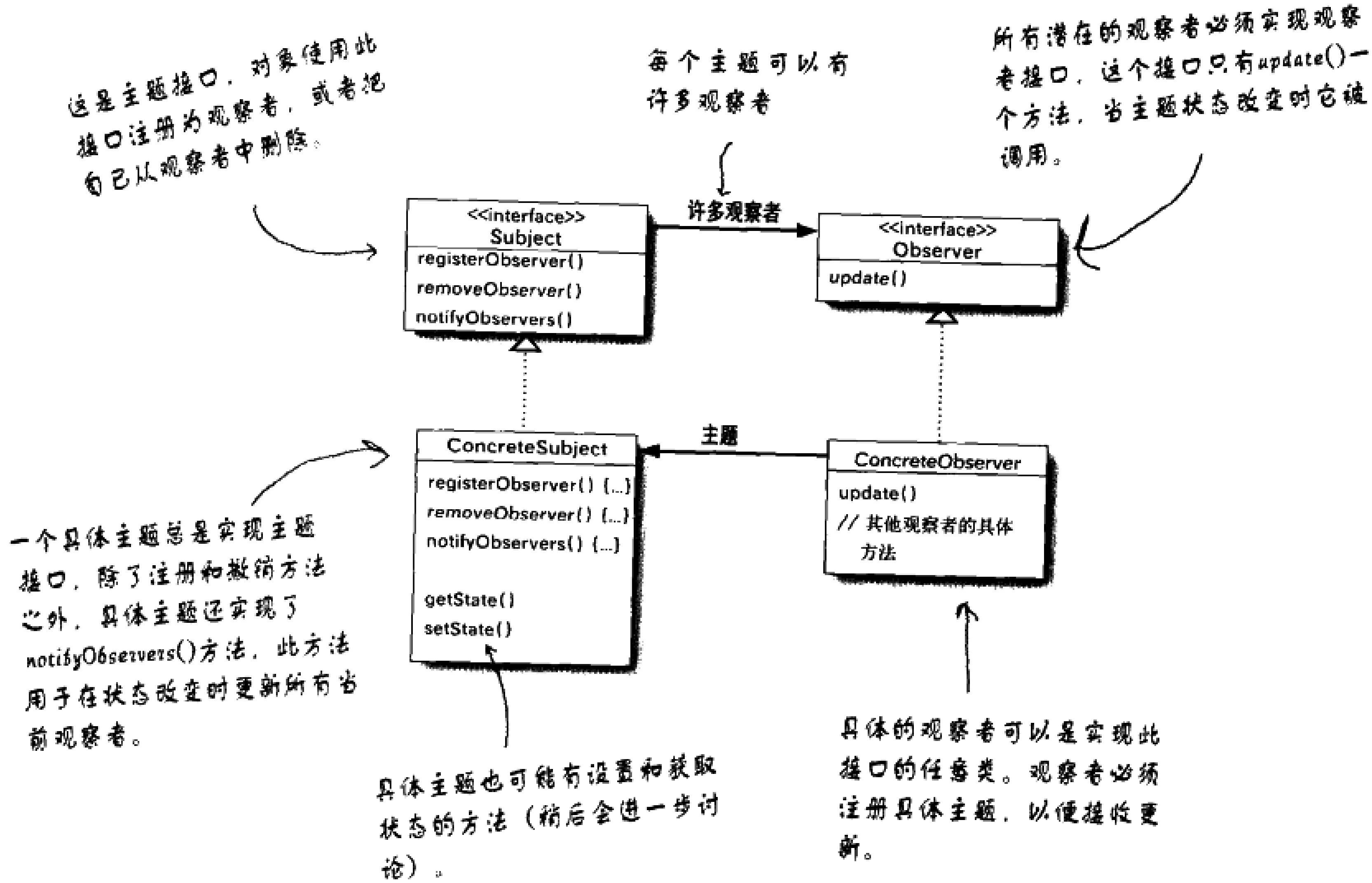
稍后你会看到，实现观察者模式的方法不只一种，但是以包含Subject与Observer接口的类设计的做法最常见。

让我们快来看看吧……

观察者模式定义了一系列对象之间的一对多关系。

当一个对象改变状态，其他依赖者都会收到通知。

定义观察者模式：类图



there are no Dumb Questions

问： 这和一对多的关系有何关联？

答： 利用观察者模式，主题是具有状态的对象，并且可以控制这些状态。也就是说，有"一个"具有状态的主题。另一方面，观察者使用这些状态，虽然这些状态并不属于他们。有许多的观察者，依赖主题来告诉他们状态何时改变了。这就产生一个关系："一个"主题对"多个"观察者的关系。

问： 其间的依赖是如何产生的？

答： 因为主题是真正拥有数据的人，观察者是主题的依赖者，在数据变化时更新，这样比起让许多对象控制同一份数据来，可以得到更干净的OO设计。

松耦合的威力

当两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节。

观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

为什么呢？

关于观察者的一切，主题只知道观察者实现了某个接口（也就是Observer接口）。主题不需要知道观察者的具体类是谁、做了些什么或其他任何细节。

任何时候我们都可以增加新的观察者。因为主题唯一依赖的东西是一个实现Observer接口的对象列表，所以我们可以随时增加观察者。事实上，在运行时我们可以用新的观察者取代现有的观察者，主题不会受到任何影响。同样的，也可以在任何时候删除某些观察者。

你能够找到多少种不同的改变？

有新类型的观察者出现时，主题的代码不需要修改。假如我们有个新的具体类需要当观察者，我们不需要为了兼容新类型而修改主题的代码，所有要做的就是在新的类里实现此观察者接口，然后注册为观察者即可。主题不在乎别的，它只会发送通知给所有实现了观察者接口的对象。

我们可以独立地复用主题或观察者。如果我们在其他地方需要使用主题或观察者，可以轻易地复用，因为二者并非紧耦合。

改变主题或观察者其中一方，并不会影响另一方。因为两者是松耦合的，所以只要他们之间的接口仍被遵守，我们就可以自由地改变他们。

设计原则

为了交互对象之间的松耦合设计而努力。

松耦合的设计之所以能让我们建立有弹性的OO系统，能够应对变化，是因为对象之间的互相依赖降到了最低。

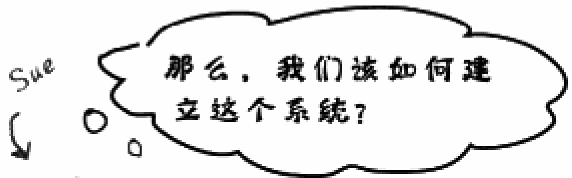
Sharpen your pencil

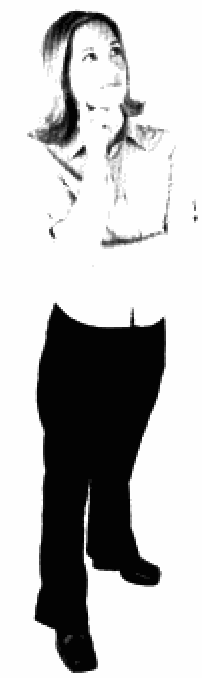
在继续后面的内容之前，请试着画出实现气象站所需要的类，其中包括WeatherData类及布告板组件。确定你的图能够显示出各个部分如何结合起来，以及别的开发人员如何能够实现他自己的布告板组件。

如果你需要一点小帮助，请阅读下一页，你的队友正在讨论如何设计气象站。

办公室隔间对话

回到气象站项目，你的队友们已经开始全面思考这个问题了……





Mary：这个嘛！使用观察者模式啰！

Sue：是的……但是如何应用？

Mary：唔，我们再看一下定义好了：

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

Mary：当你思考这个定义时，你会发现很有道理。我们的WeatherData类正是此处所说的“一”，而我们的“多”正是使用天气观测的各种布告板。

Sue：没错。WeatherData对象的确是有状态，包括了温度、湿度、气压，而这些值都会改变。

Mary：对呀！而且，当这些观测值改变时，必须通知所有的布告板，好让它们各自做出处理。

Sue：好棒！我现在知道如何将观察者模式应用在气象站问题上了。

Mary：还有一些问题有待理清，我现在还不太了解它们的解决方法。

Sue：什么问题？

Mary：其中一个问题是，我们如何将气象观测值放到布告板上。

Sue：回头去看看观察者模式的图，如果我们把WeatherData对象当作主题，把布告板当作观察者，布告板为了取得信息，就必须先向WeatherData对象注册。对不对？

Mary：是的……一旦WeatherData知道有某个布告板的存在，就会适时地调用布告板的某个方法来告诉布告板观测值是多少。

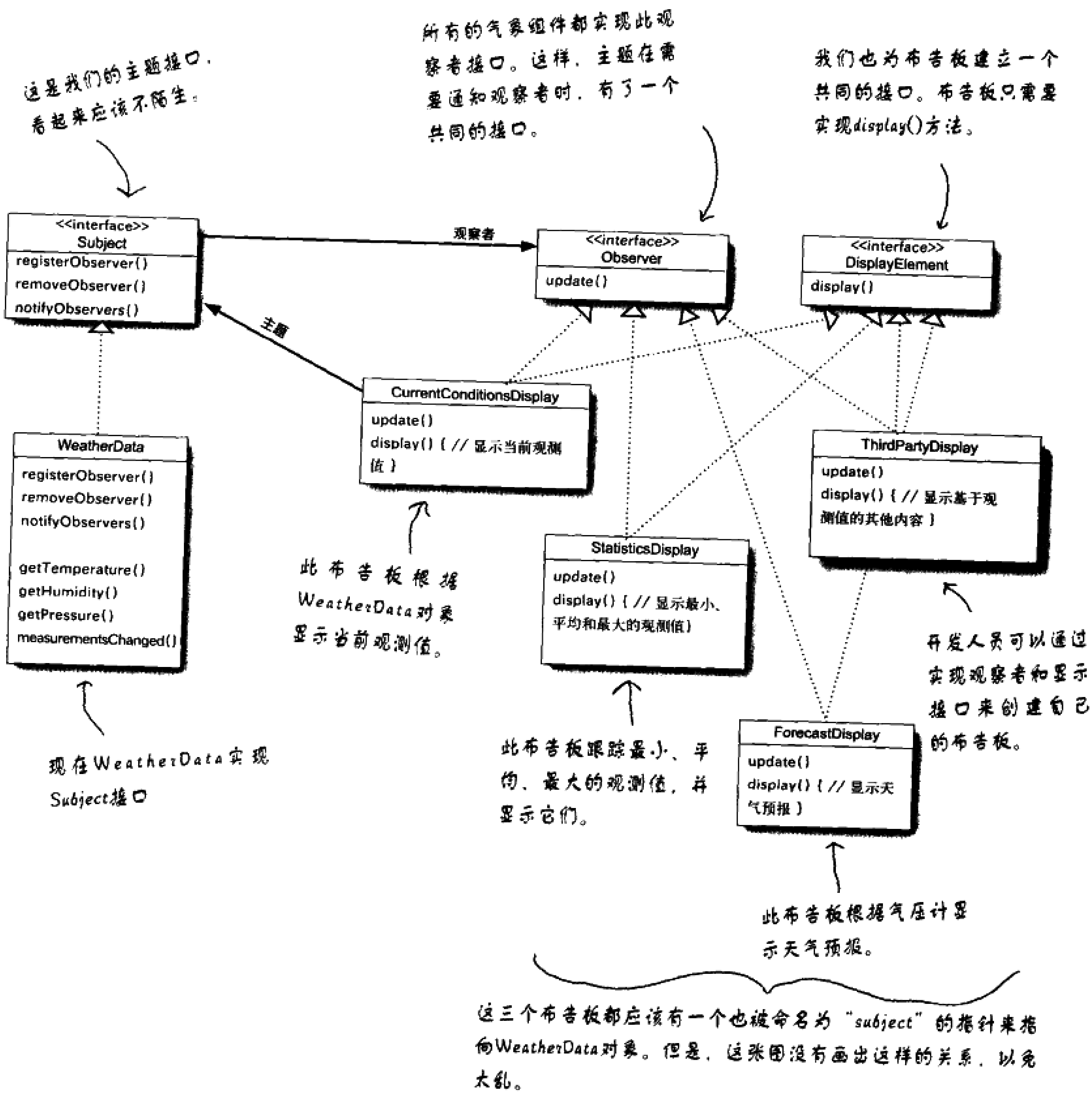
Sue：我们必须记得，每个布告板都有差异，这也就是为什么我们需要一个共同的接口的原因。尽管布告板的类都不一样，但是它们都应该实现相同的接口，好让WeatherData对象能够知道如何把观测值送给它们。

Mary：我懂你的意思。所以每个布告板都应该有一个大概名为update()的方法，以供WeatherData对象调用。

Sue：而这个update()方法应该在所有布告板都实现的共同接口里定义。

设计气象站

看看这个设计图，和你的设计图有何异同？



实现气象站

依照两页前Mary和Sue的讨论，以及上一页的类图，我们要开始实现这个系统了。稍后，你将会在本章看到Java为观察者模式提供了内置的支持，但是，我们暂时不用它，而是先自己动手。虽然，某些时候可以利用Java内置的支持，但是有许多时候，自己建立这一切会更具弹性（况且建立这一切并不是很麻烦）。所以，让我们从建立接口开始吧：

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

这两个方法都需要一个观察者作为变量，该观察者是用来注册或被删除的。

当主题状态改变时，这个方法会被调用，以通知所有的观察者。

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

当气象观测值改变时，主题会把这些状态值当作方法的参数，传送给观察者。

所有的观察者都必须实现update()方法，以实现观察者接口。在这里，我们按照Mary和Sue的想法把观测值传入观察者中。

```
public interface DisplayElement {
    public void display();
}
```

DisplayElement接口只包含了一个方法，也就是display()。当布告板需要显示时，调用此方法。

BRAIN POWER

Mary和Sue认为：把观测值直接传入观察者中是更新状态的最直接的方法。你认为这样的做法明智吗？暗示：这些观测值的种类和个数在未来有可能改变吗？如果以后会改变，这些变化是否被很好地封装？或者是需要修改许多代码才能办到？

关于将更新的状态传送给观察者，你能否想到更好的方法解决此问题？

别担心，在我们完成第一次实现后，我们会再回来探讨这个设计决策。

在WeatherData中实现主题接口

提醒你：为了节省篇幅，我们在代码中没有列出import和package语句。你可以到wickedlysmart网站找到完整的源代码，URL在本书的第xxxv页。

还记得我们在本章一开始的地方就试图实现WeatherData类吗？你可以去回顾一下。现在，我们要用观察者模式实现……

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // WeatherData的其他方法
}
```

WeatherData现在实现了Subject接口。

我们加上一个ArrayList来纪录观察者，此ArrayList是在构造器中建立的。

当注册观察者时，我们只要把它加到ArrayList的后面即可。

同样地，当观察者想取消注册，我们把它从ArrayList中删除即可。

这部分是Subject接口的实现。

有趣的地方来了！在这里，我们把状态告诉每一个观察者。因为观察者都实现了update()，所以我们知道如何通知它们。

当从气象站得到更新观测值时，我们通知观察者。

我们想要每本书随书赠送一个小型气象站，但是出版社不肯。所以，和从装置中读取实际的气象数据相比，我们宁愿利用这个方法来测试布告板。或者，为了好玩，你也可以写代码从网站上抓取观测值。

现在，我们来建立布告板吧！

我们已经把WeatherData类写出来了，现在轮到布告板了。Weather-O-Rama气象站订购了三个布告板：目前状况布告板、统计布告板和预测布告板。我们先看看目前状况布告板。一旦你熟悉此布告板之后，可以在本书的代码目录中，找到另外两个布告板的源代码，你会觉得这些布告板都很类似。

此布告板实现了Observer接口，所以可以从WeatherData对象中获得改变。

它也实现了DisplayElement接口，因为我们的API规定所有的布告板都必须实现此接口。

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

构造器需要 weatherData对象（也就是主题）作为注册之用。

当update()被调用时，我们把温度和湿度保存起来，然后调用display()。

display()方法就只是把最近的温度和湿度显示出来。

---

there are no Dumb Questions

问： update()是最适合调用display()的地方吗？

答： 在这个简单的例子中，当值变化的时候调用display()，是很合理的。然而，你是对的，的确是有很多更好的方法来设计显示数据的方式。当我们谈到MVC（Model-View-Controller）模式时会再作说明。

问： 为什么要保存对Subject的引用呢？构造完后似乎用不着了呀？

答： 的确如此，但是以后我们可能想要取消注册，如果已经有了对Subject的引用会比较方便。

启动气象站



❶ 先建立一个测试程序

气象站已经完成得差不多了，我们还需要一些代码将这一切连接起来。这是我们的第一次尝试，本书中稍后我们会再回来确定每个组件都能通过配置文件来达到容易“插拔”。现在开始测试吧：

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

首先，建立一个WeatherData对象。

如果你还不想下载完整的代码，可以将这两行注释掉，就能顺利执行了。

建立三个布告板，并把WeatherData对象传给它们。

模拟新的气象测量。

❷ 运行程序，让观察者模式表演魔术。

File Edit Window Help StormyWeather

```
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Sharpen your pencil

Johnny Hurricane（Weather-O-Rama气象站的CEO）刚刚来电告知，他们还需要酷热指数（HeatIndex）布告板，这是不可或缺的。细节如下：

酷热指数是一个结合温度和湿度的指数，用来显示人的温度感受。可以利用温度T和相对湿度RH套用下面的公式来计算酷热指数：

heatindex =

$$16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3$$

开始练习打字吧！

开玩笑的啦！别担心，你不需要亲自输入此公式，只要建立你自己的HeatIndexDisplay.java文件并把公式从heatindex.txt文件中拷贝进来就可以了。

heatindex.txt文件可从wickedlysmart.com取得

这个公式是怎么回事？你可以参考《Head First气象学》，或者问问国家气象局的员工

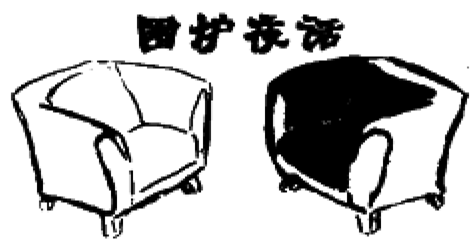
（或用Google搜索）。

当你完成后，输出结果应如下所示：

输出结果有改变的地方在这里。

File Edit Window Help OverdaRainbow

```
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```



今夜话题：主题和观察者就使观察者获得状态信息的正确方法发生了争吵。

主题 / **观察者**

主题： 我很高兴，我们终于有机会面对面聊天了。

观察者： 是这样吗？我以为你根本不在乎我们这群观察者呢。

主题： 唉呀！我把该做的事都做到了，不是吗？我总是会通知你们发生什么事了……我虽然不知道你们是谁，但这不意味着我不在乎你们。况且，我知道关于你们的一件重要的事：你们实现了Observer接口。

观察者： 是呀，但这只是关于我的一小部分罢了！无论如何，我对你更了解……

主题： 是吗？说来听听！

观察者： 嗯！你总是将你的状态传给我们，所以我们可以知道你内部的情况。有时候，这很烦人的……

主题： 拜托，我必须主动送出我的状态和通知给大家，好让你们这些懒惰的观察者知道发生什么事了。

观察者： 咳！等等。我说主题先生，首先，我们并不懒，在你那些“很重要”通知的空档中，我们还有别的事要做。另外，为何由你主动送数据过来，而不是让我们主动去向你索取数据？

主题： 嗯……这样或许也行，只是我必须因此门户大开，让你们全都可以进来取得你们需要的状态，这样太危险了。我不能让你们进来里面大肆挖掘我的各种数据。

主题

观察者

观察者：你何不提供一些公开的getter方法，让我们“拉”走我们需要的状态？

主题：是的，我可以让你们“拉”走我的状态，但是你不觉得这样对你们反而不方便吗？如果每次想要数据时都来找我，你可能要调用很多次才能收集齐全你所要的状态。这就是为什么我更喜欢“推”的原因，你们可以在一次通知中一口气得到所有东西。

观察者：死鸭子嘴硬！观察者种类这么多，你不可能事先料到我们每个人的需求，还是让我们直接去取得我们需要的状态比较恰当，这样一来，如果我们有人只需要一点点数据，就不会被强迫收到一堆数据。这么做同时也可以在以后比较容易修改。比方说，哪一天你决定扩展功能，新增更多的状态，如果采用我建议的方式，你就不用修改和更新对每位观察者的调用，只需改变自己来允许更多的getter方法来取得新增的状态。

主题：是的。两种做法都有各自的优点。我注意到Java内置的Observer 模式两种做法都支持。

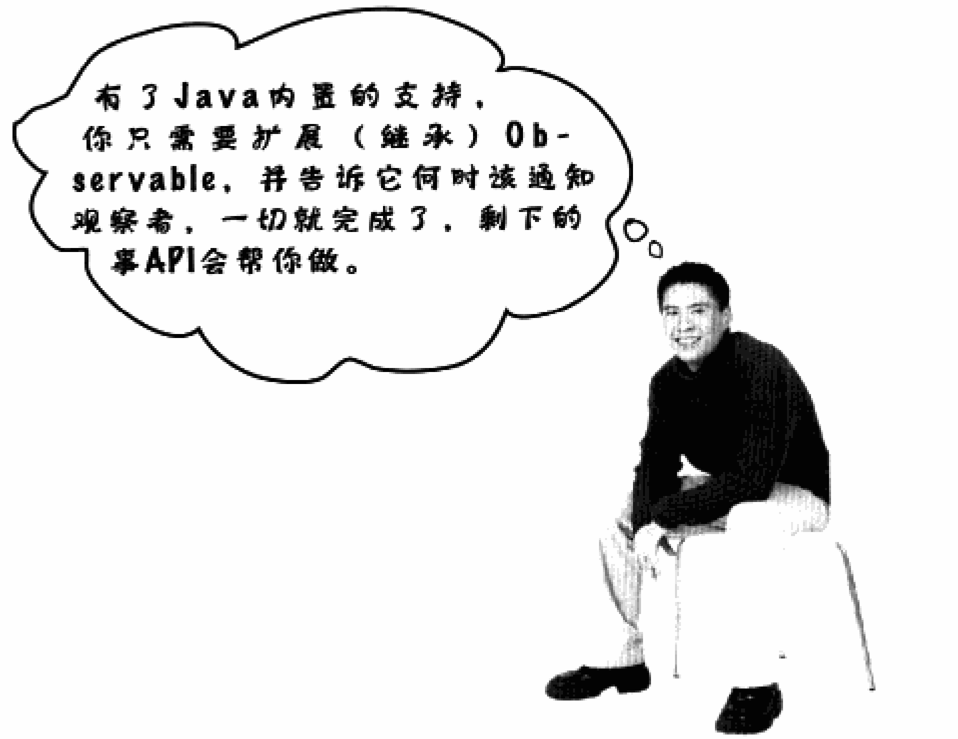
观察者：真的吗？我们得去瞧瞧……

主题：太好了，或许我会看到一个“拉”的好例子，因而改变我的想法。

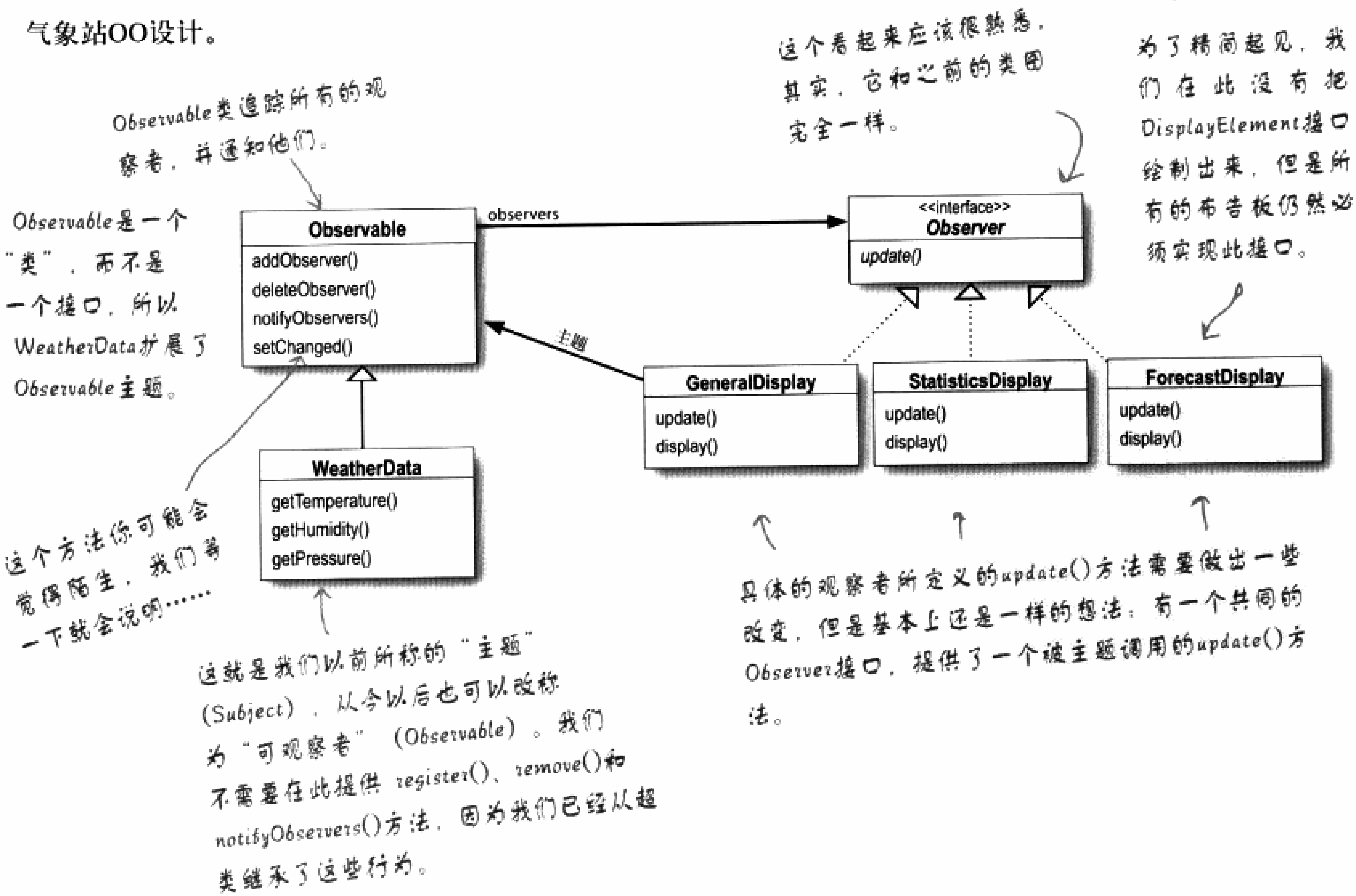
观察者：什么？我们会有意见相同的一天？不会吧！

使用Java内置的观察者模式

到目前为止，我们已经从无到有地完成了观察者模式，但是，Java API有内置的观察者模式。java.util包（package）内包含最基本的Observer接口与Observable类，这和我们的Subject接口与Observer接口很相似。Observer接口与Observable类使用上更方便，因为许多功能都已经事先准备好了。你甚至可以使用推（push）或拉（pull）的方式传送数据，稍后就会看到这样的例子。



为了更了解java.uitl.Observer和java.util.Observable，看看下面的图，这是修改后的气象站OO设计。



Java内置的观察者模式如何运作

Java内置的观察者模式运作方式，和我们在气象站中的实现类似，但有一些小差异。最明显的差异是WeatherData（也就是我们的主题）现在扩展自Observable类，并继承到一些增加、删除、通知观察者的方法（以及其他的方法）。Java版本的用法如下：

如何把对象变成观察者……

如同以前一样，实现观察者接口（java.uitl.Observer），然后调用任何Observable对象的addObserver()方法。不想再当观察者时，调用deleteObserver()方法就可以了。

可观察者要如何送出通知……

首先，你需要利用扩展java.util.Observable接口产生“可观察者”类，然后，需要两个步骤：

1. 先调用setChanged()方法，标记状态已经改变的事实。

2. 然后调用两种notifyObservers()方法中的一个：

当通知时，此版本可以传送任何的数据对象给每一个观察者。

`notifyObservers()` 或 `notifyObservers(Object arg)`

观察者如何接收通知……

同以前一样，观察者实现了更新的方法，但是方法的签名不太一样：

data object

`update(Observable o, Object arg)`

主题本身当作第一个变量，好让观察者知道是哪个主题通知它的。

这正是传入notifyObservers()的数据对象。如果没有说明则为空。

如果你想“推”（push）数据给观察者，你可以把数据当作数据对象传送给notifyObservers(arg)方法。否则，观察者就必须从可观察者对象中“拉”（pull）数据。如何拉数据？我们再做一遍气象站，你很快就会看到。



setChanged()方法用来标记状态已经改变的事实，好让notifyObservers()知道当它被调用时应该更新观察者。如果调用notifyObservers()之前没有先调用setChanged()，观察者就“不会”被通知。让我们看看Observable 内部，以了解这一切：

幕后花絮

Observable类的伪代码

```
setChanged() {
   changed = true
}

notifyObservers(Object arg) {
   if (changed) {
      for every observer on the list {
         call update (this, arg)
      }
      changed = false
   }
}

notifyObservers() {
   notifyObservers(null)
}
```

setChanged()方法把changed标志设为true。

notifyObservers() 只会在changed标为“true”时通知观察者。

在通知观察者之后，把changed标志设回false。

这样做有其必要性。setChanged()方法可以让你在更新观察者时，有更多的弹性，你可以更适当地通知观察者。比方说，如果没有setChanged()方法，我们的气象站测量是如此敏锐，以致于温度计读数每十分之一度就会更新，这会造成WeatherData对象持续不断地通知观察者，我们并不希望看到这样的事情发生。如果我们希望半度以上才更新，就可以在温度差距到达半度时，调用setChanged()，进行有效的更新。

你也许不会经常用到此功能，但是把这样的功能准备好，当需要时马上就可以使用。总之，你需要调用setChanged()，以便通知开始运转。如果此功能在某些地方对你有帮助，你可能也需要clearChanged()方法，将changed状态设置回false。另外也有一个hasChanged()方法，告诉你changed标志的当前状态。

利用内置的支持重做气象站

首先，把WeatherData改成使用java.util.Observable

❶ 记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在继承Observable。

❸ 我们不再需要追踪观察者了，也不需要管理注册与删除（让超类代劳即可）。所以我们把注册、添加、通知的相关代码删除。

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers(); ★
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

❹ 我们的构造器不再需要为了记住观察者们而建立数据结构了。

★ 注意：我们没有调用notifyObservers()传送数据对象，这表示我们采用的做法是“拉”。

❺ 在调用notifyObservers()之前，要先调用setChanged()来指示状态已经改变。

❻ 这些并不是新方法，只是因为我们要使用“拉”的做法，所以才提醒你有这些方法。察者会利用这些方法取得WeatherData对象的状态。

现在，让我们重做CurrentConditionsDisplay

❶ 再说一遍，记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在实现java.util.Observer接口。

❸ 现在构造器需要一Observable当参数，并将CurrentCondi-tionsDisplay对象登记成为观察者。

❹ 改变update()方法，增加Observable和数据对象作为参数。

❺ 在update()中，先确定可观察者属于WeatherData类型，然后利用getter方法获取温度和湿度测量值，最后调用display()。

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

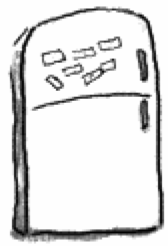
    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```



代码帖



ForecastDisplay类的代码小纸片在冰箱上被弄乱了。你能够重新排列它们，好恢复原来的样子吗？有些大括号掉到地上了，因为太小捡起来不易，所以如果你觉得需要大括号时，可以自行加上。

```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // 这里显示代码
}
```

```
lastPressure = currentPressure;
currentPressure = weatherData.getPressure();
```

```
private float currentPressure = 29.92f;
private float lastPressure;
```

```
WeatherData weatherData =
(WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

运行新的代码

让我们运行新的代码，以确定它是对的……

```
File Edit Window Help TryTihisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```

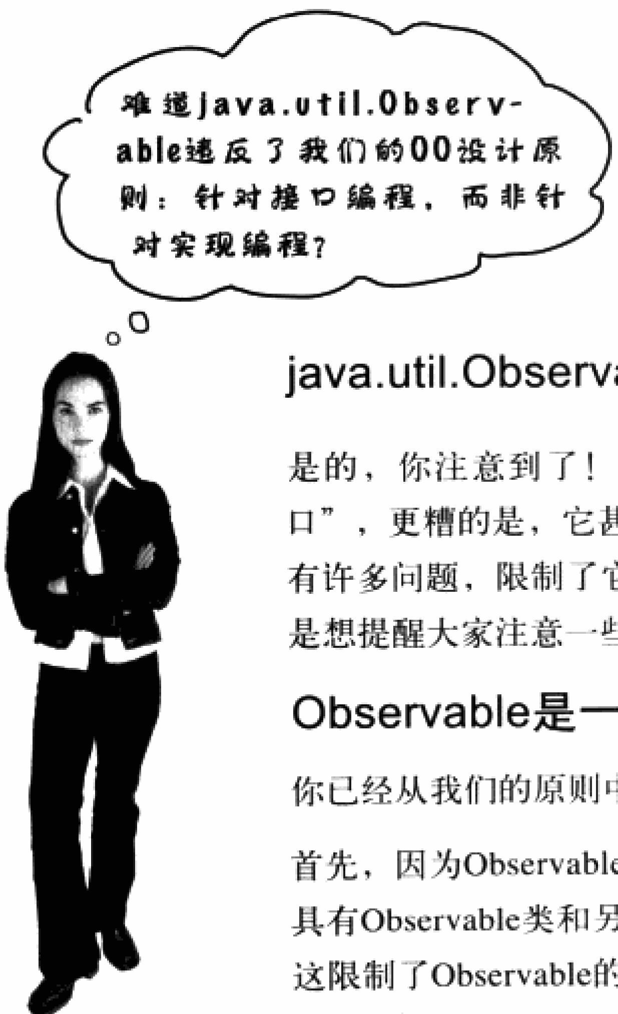
嗯！你注意到差别了吗？再看一次……

你会看到相同的计算结果，但是奇怪的地方在于，文字输出的次序不一样。怎么会这样呢？在继续之前，请花一分钟的时间思考……

不要依赖于观察者被通知的次序

java.uitl.Observable实现了它的notifyObservers()方法，这导致了通知观察者的次序不同于我们先前的次序。谁也没有错，只是双方选择不同的方式实现罢了。

但是可以确定的是，如果我们的代码依赖这样的次序，就是错的。为什么呢？因为一旦观察者/可观察者的实现有所改变，通知次序就会改变，很可能就会产生错误的结果。这绝对不是我们所认为的松耦合。



java.util.Observable的黑暗面

是的，你注意到了！如同你所发现的，可观察者是一个“类”而不是一个“接口”，更糟的是，它甚至没有实现一个接口。不幸的是，java.util.Observable的实现有许多问题，限制了它的使用和复用。这并不是说它没有提供有用的功能，我们只是想提醒大家注意一些事实。

Observable是一个类

你已经从我们的原则中得知这不是一件好事，但是，这到底会造成什么问题呢？

首先，因为Observable是一个“类”，你必须设计一个类继承它。如果某类想同时具有Observable类和另一个超类的行为，就会陷入两难，毕竟Java不支持多重继承。这限制了Observable的复用潜力（而增加复用潜力不正是我们使用模式最原始的动机吗？）。

再者，因为没有Observable接口，所以你无法建立自己的实现，和Java内置的Observer API搭配使用，也无法将java.util的实现换成另一套做法的实现（比方说，

Observable将关键的方法保护起来

如果你看看Observable API，你会发现setChanged()方法被保护起来了（被定义成protected）。那又怎么样呢？这意味着：除非你继承自Observable，否则你无法创建Observable实例并组合到你自己的对象中来。这个设计违反了第二个设计原则：“多用组合，少用继承”。

做什么呢？

如果你能够扩展java.util.Observable，那么Observable“可能”可以符合你的需求。否则，你可能需要像本章开头的做法那样自己实现这一整套观察者模式。不管用哪一种方法，反正你都已经熟悉观察者模式了，应该都能善用它们。

在JDK中，还有哪些地方可以找到观察者模式

在JDK中，并非只有在java.util中才能找到观察者模式，其实在JavaBeans和Swing中，也都实现了观察者模式。现在，你已经具备足够的能力来自行探索这些API，但是我们还是在此稍微提一个简单的Swing例子，让你感受一下其中的乐趣。

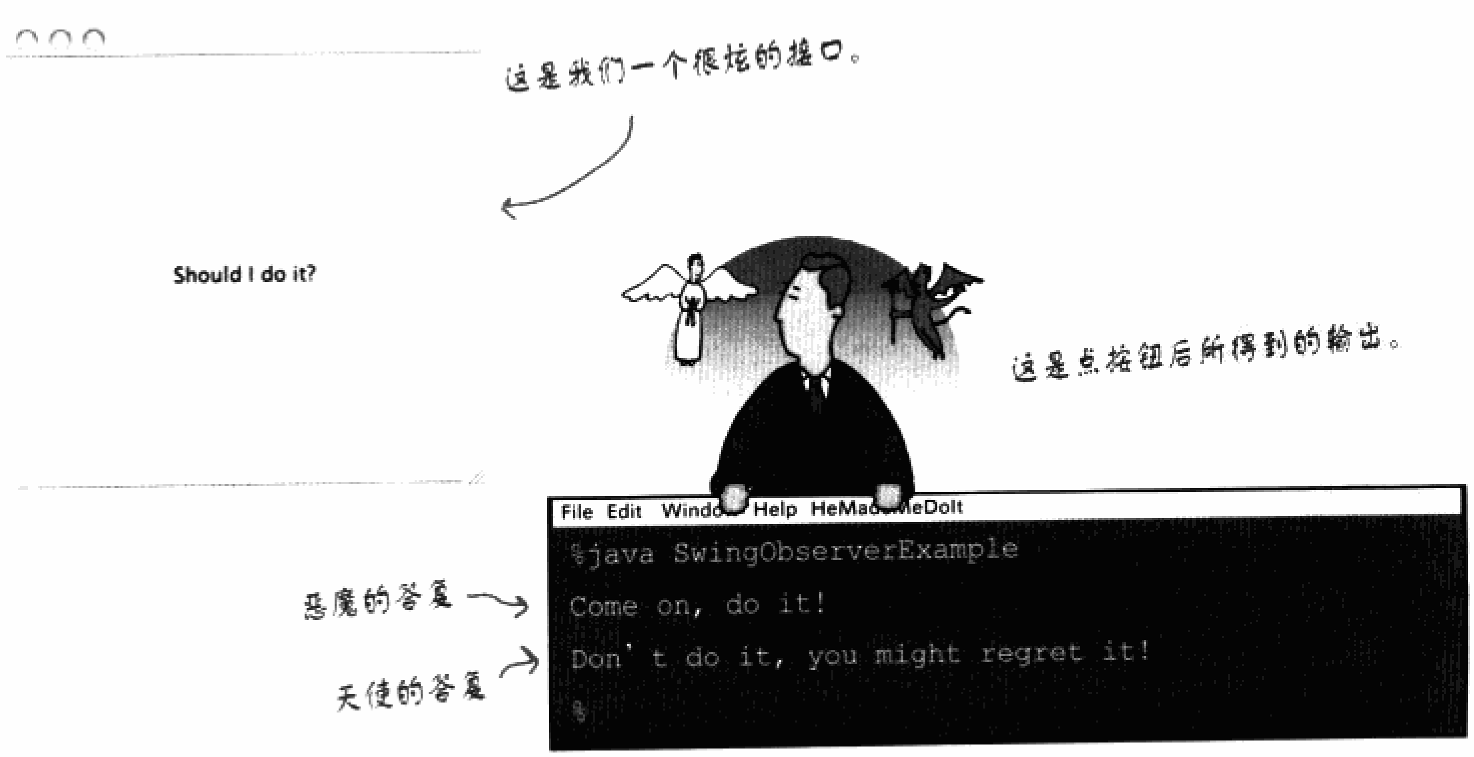
如果你对JavaBeans里的观察者模式感到好奇，可以查一下PropertyChangeListener接口。

背景介绍……

让我们看看一个简单的Swing API：JButton。如果你观察一下JButton的超类AbstractButton，会看到许多增加与删除倾听者（listener）的方法，这些方法可以让观察者感应到Swing组件的不同类型事件。比方说：ActionListener让你“倾听”可能发生在按钮上的动作，例如按下按钮。你可以在Swing API中找到许多不同类型的倾听者。

一个小的、改变生活的程序

我们的程序很简单，你有一个按钮，上面写着“Should I do it?”（我该做吗？）。当你按下按钮，倾听者（观察者）必须回答此问题。我们实现了两个倾听者，一个是天使（AngelListener），一个是恶魔（DevilListener）。程序的行为如下：



代码是这样的……

这个改变生活的程序需要的代码很短。我们只需要建立一个JButton对象，把它加到JFrame，然后设置好倾听者就行了。我们打算用内部类（inner class）作为倾听者类（这样的技巧在Swing中很常见）。如果你对内部类或Swing不熟悉，可以读一读《Head First Java》中的并于“获得GUI”的章节。

简单的Swing应用：建立一个JFrame，然后放上一个按钮。

```
public class SwingObserverExample {
    JFrame frame;

    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
        frame.getContentPane().add(BorderLayout.CENTER, button);
        // 在这里设置frame属性
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

制造出两个倾听者（观察者），一个天使，一个恶魔。

这是观察者的类定义，定义成内部类（你也可以不这么做）。

当主题（JButton）的状态改变时，在本例中，不是调用update()，而是调用actionPerformed()。



设计箱内的工具

欢迎来到第2章的结尾，你的对象工具箱内又多了一些东西……

OO基础

抽象

OO原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

为交互对象之间的松耦合设计而努力

这是你的新原则。请牢记，松耦合设计更有弹性，更能应对变化。

OO模式

策略……

来，……

让算……

观察者模式——在对象之间定义一对多的依赖，这样一来，当一个对象改变状态，依赖它的对象都会收到通知，并自动更新。

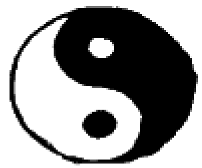
一个新的模式，以松耦合方式在一系列对象之间沟通状态。我们目前还没看到观察者模式的代表人物——MVC，以后就会看到了。

要点

- 观察者模式定义了对象之间一对多的关系。
- 主题（也就是可观察者）用一个共同的接口来更新观察者
- 观察者和可观察者之间用松耦合方式结合（loosecoupl-ing），可观察者不知道观察者的细节，只知道观察者实现了观察者接口。
- 使用此模式时，你可从被观察者处推（push）或拉（pull）数据（然而，推的方式被认为更“正确”）。
- 有多个观察者时，不可以依赖特定的通知次序。
- Java有多种观察者模式的实现，包括了通用的java.util.Observable。
- 要注意java.util.Observable实现上所带来的一些问题。
- 如果有必要的话，可以实现自己的Observable，这并不难，不要害怕。
- Swing大量使用观察者模式，许多GUI框架也是如此。
- 此模式也被应用在许多地方，例如：JavaBeans、RMI。



练习



挑战设计原则

对于每一个设计原则，请描述观察者模式如何遵循此原则。

设计原则

找出程序中会变化的方面，然后将其和固定不变的方面相分离。

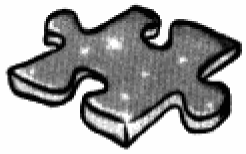
设计原则

针对接口编程，不针对实现编程

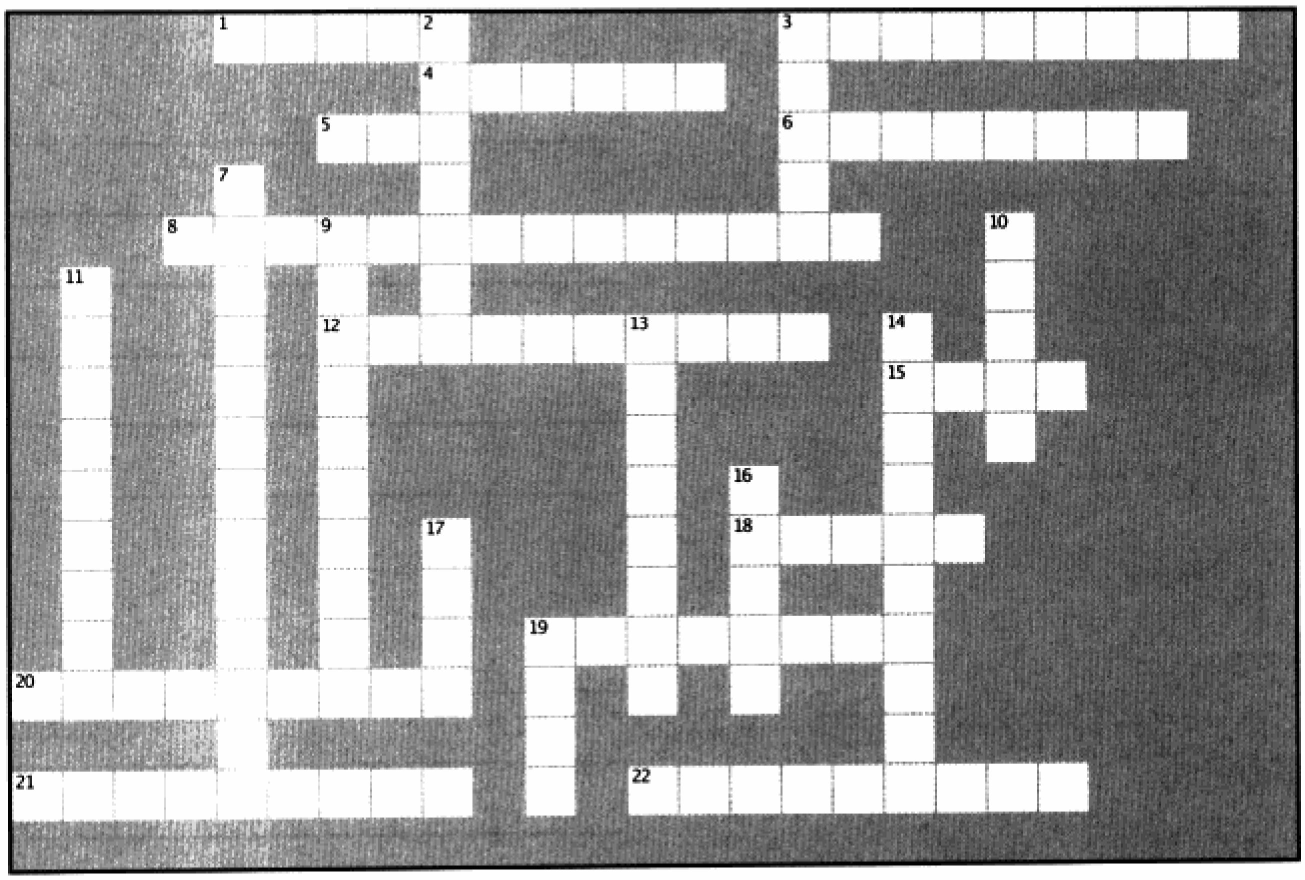
这一个比较难回答。给一点暗示：想想看观察者和主题是如何搭配工作的。

设计原则

多用组合，少用继承



再次为你的右脑找些事情做吧！
这次所有的词都来自第2章。



横排提示：

1. Observable is a ______ not an interface
3. Devil and Angel are _______ to the button
4. Implement this method to get notified
5. Jill got one of her own
6. CurrentConditionsDisplay implements this interface
8. How to get yourself off the Observer list
12. You forgot this if you're not getting notified when you think you should be
15. One Subject likes to talk to _____ observers
18. Don't count on this for notification
19. Temperature, humidity and _______
20. Observers are ________ on the Subject
21. Program to an _______ not an implementation
22. A Subject is similar to a __________

竖排提示：

2. Ron was both an Observer and a __________
3. You want to keep your coupling ________
7. He says you should go for it
9. _____ can manage your observers for you
10. Java framework with lots of Observers
11. Weather-O-Rama's CEO named after this kind of storm
13. Observers like to be _________ when something new happens
14. The WeatherData class _________ the Subject interface
16. He didn't want any more ints, so he removed himself
17. CEO almost forgot the ______ index display
19. Subject initially wanted to ____ all the data to Observer

习题解答

Sharpen your pencil

在我们的第一个实现中，下列哪种说法正确？（多选）

- ☑ A 我们是针对具体实现编程，而非针对接口。
- ☑ B. 对于每个新的布告板，我们都得修改代码。
- ☑ C. 我们无法在运行时动态地增加或删除布告板。
- ☐ D. 布告板没有实现一个共同的接口。
- ☑ E. 我们尚未封装改变的部分。
- ☐ F. 我们侵犯了WeatherData类的封装。

挑战设计原则

设计原则

找出程序中会变化的方面，然后将其和固定不变的方面相分离。

在观察者模式中，会改变的是主题的状态，以及观察者的数目和类型。用这个模式，你可以改变依赖于主题状态的对象，却不必改变主题。这就叫提前规划！

设计原则

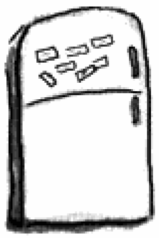
针对接口编程，不针对实现编程。

主题与观察者都使用接口：观察者利用主题的接口向主题注册，而主题利用观察者接口通知观察者。这样可以让两者之间运作正常，又同时具有松耦合的优点。

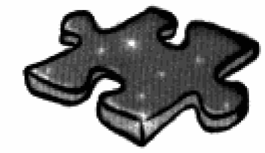
设计原则

多用组合，少用继承。

观察者模式利用“组合”将许多观察者组合进主题中。对象之间的这种关系不是通过继承产生的，而是在运行时利用组合的方式而产生的。



代码帖



习题解答

```
import java.util.Observable;
import java.util.Observer;

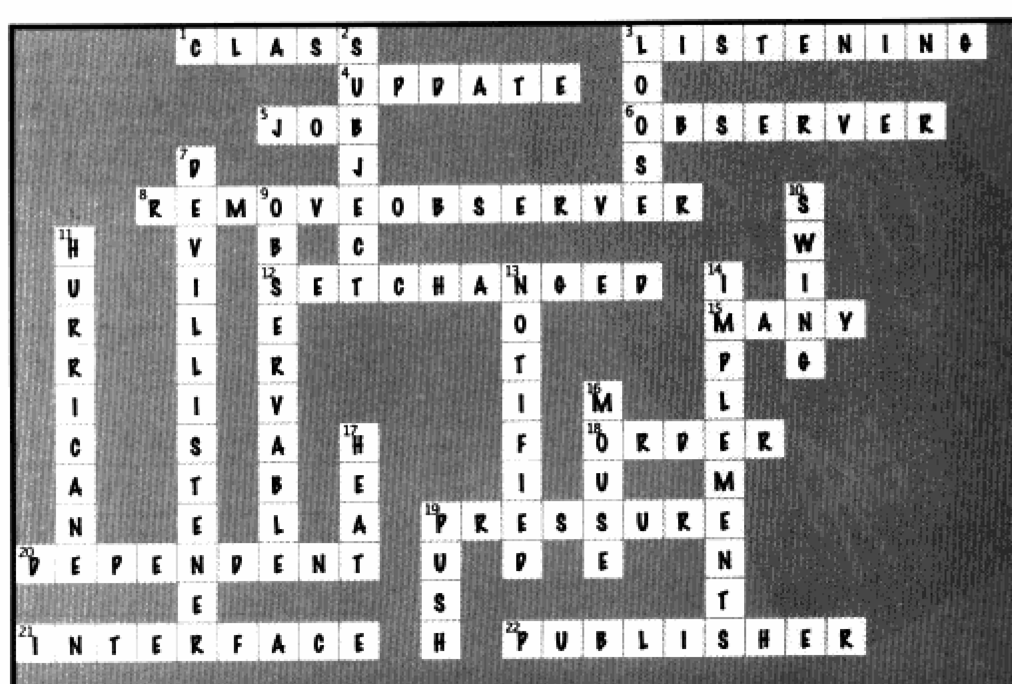
public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

    public ForecastDisplay(Observable
    observable) {
        WeatherData weatherData =
        (WeatherData)observable;
        observable.addObserver(this);
    }

    public void update(Observable observable,
        Object arg) {
        if (observable instanceof WeatherData) {
            lastPressure = currentPressure;
            currentPressure = weatherData.getPressure();
            display();
        }
    }

    public void display() {
        // 在这里显示代码
    }
}
```



3 装饰者模式

装饰对象



本章可以称为“给爱用继承的人一个全新的设计眼界”。 我们即将再度探讨典型的继承滥用问题。你将在本章学到如何使用对象组合的方式，做到在运行时装饰类。为什么呢？一旦你熟悉了装饰的技巧，你将能够在不修改任何底层代码的情况下,给你的（或别人的）对象赋予新的职责。

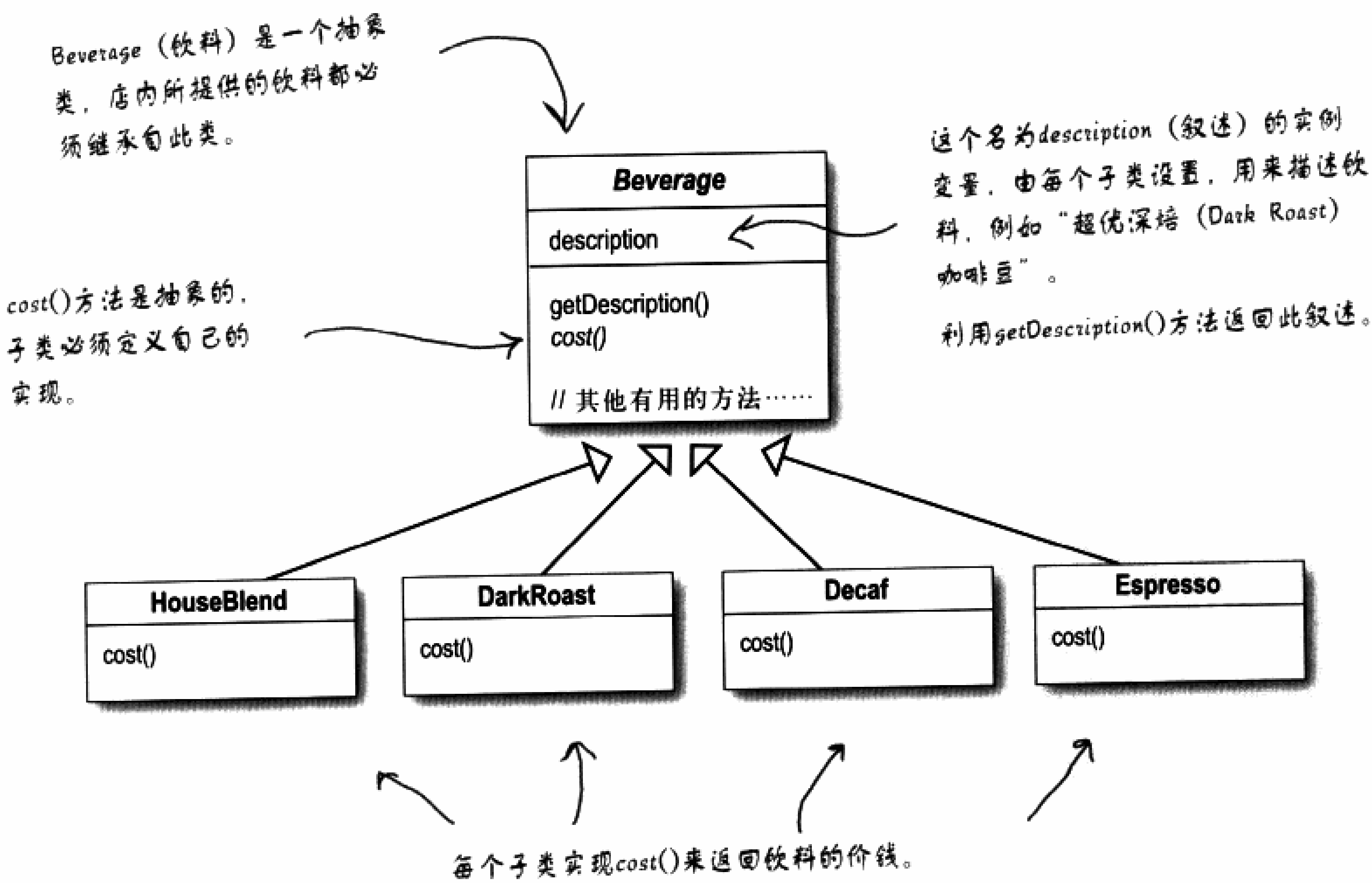
欢迎来到星巴兹咖啡



星巴兹（Starbuzz）是以扩张速度最快而闻名的咖啡连锁店。如果你在街角看到它的店，在对面街上肯定还会看到另一家。

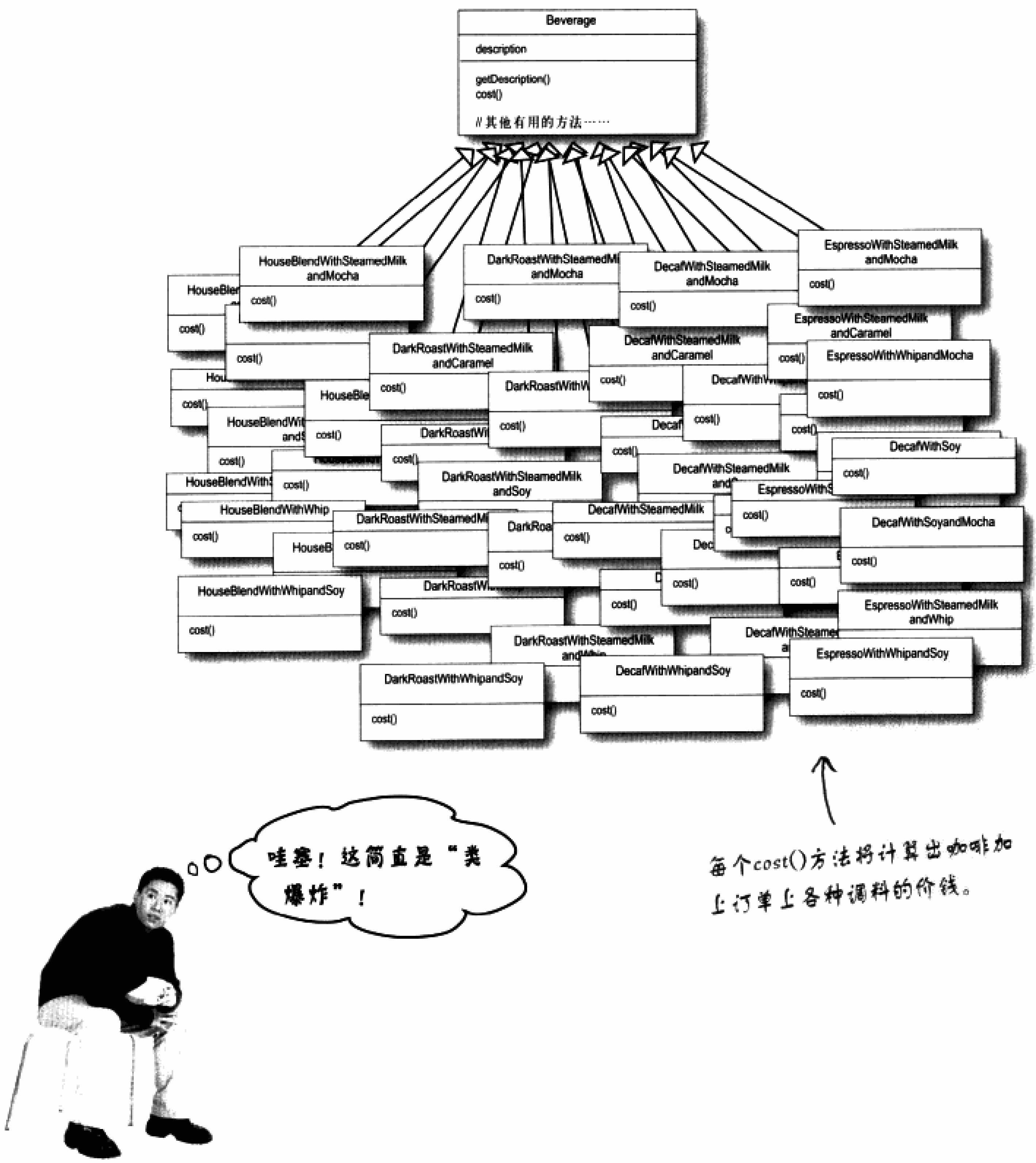
因为扩张速度实在太快了，他们准备更新订单系统，以合乎他们的饮料供应要求。

他们原先的类设计是这样的……



购买咖啡时，也可以要求在其中加入各种调料，例如：蒸奶（Steamed Milk）、豆浆（Soy）、摩卡（Mocha，也就是巧克力风味）或覆盖奶泡。星巴兹会根据所加入的调料收取不同的费用。所以订单系统必须考虑到这些调料部分。

这是他们的第一个尝试……



BRAIN POWER

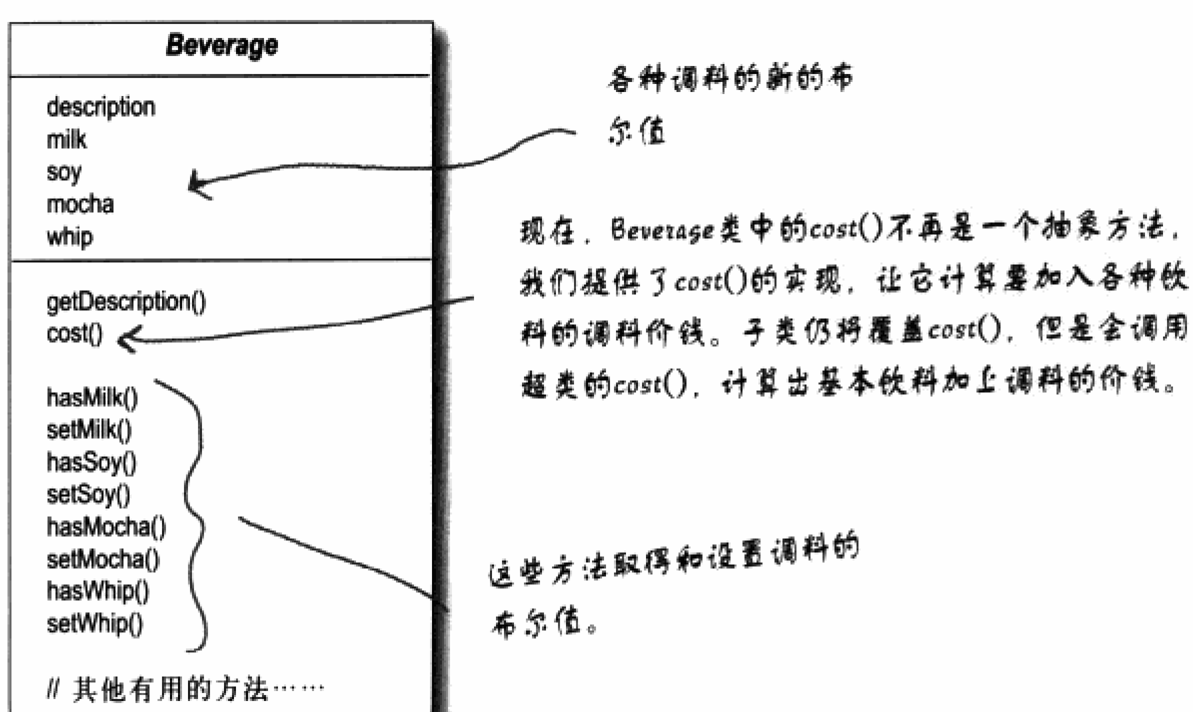
很明显，星巴兹为自己制造了一个维护恶梦。如果牛奶的价钱上扬，怎么办？新增一种焦糖调料风味时，怎么办？

造成这种维护上的困难，究竟违反了我们之前提过的哪种设计原则？

提示：违反了两个原则，而且很严重！



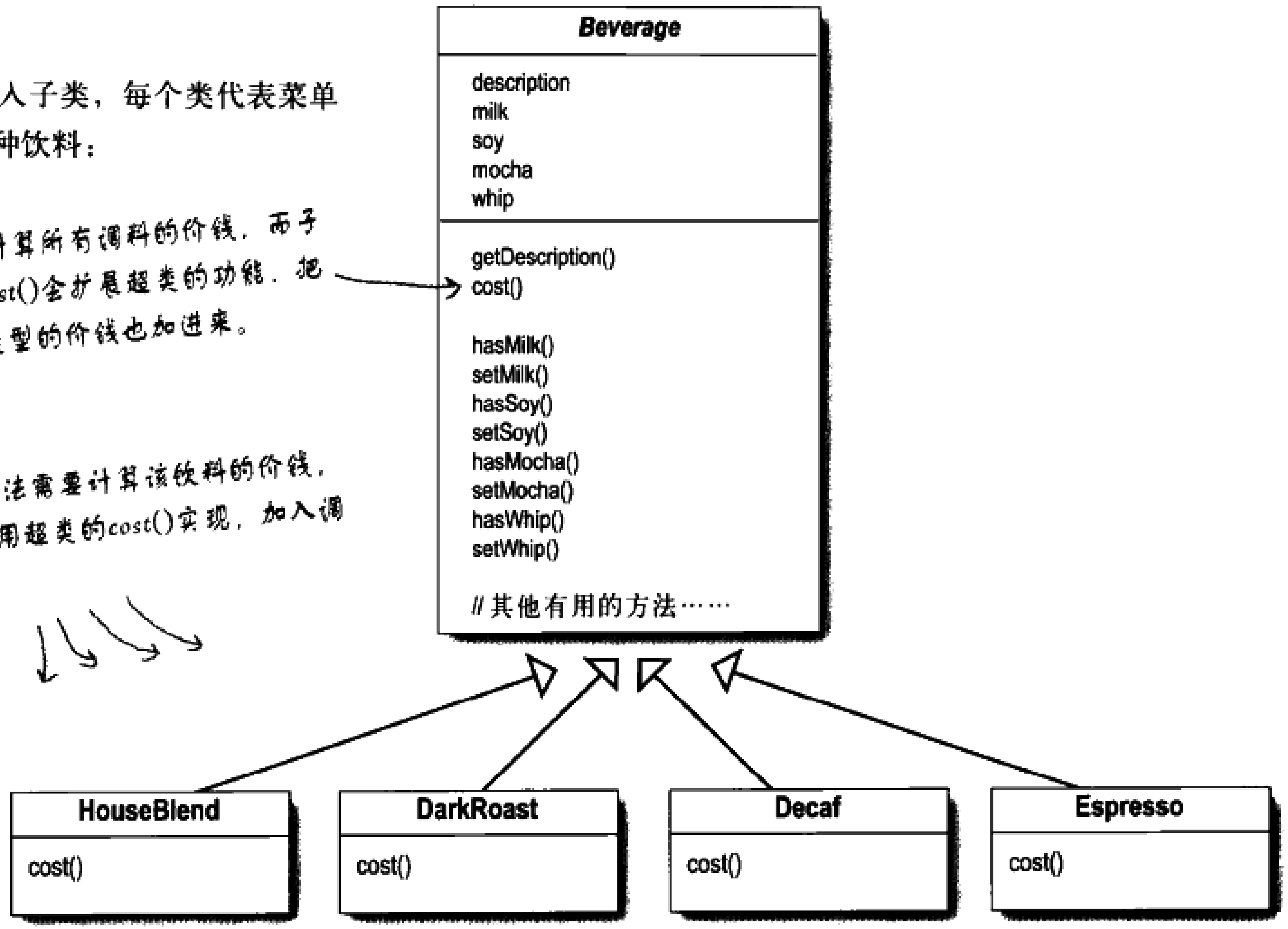
好吧！就来试试看。先从Beverage基类下手，加上实例变量代表是否加上调料（牛奶、豆浆、摩卡、奶泡……）



现在加入子类，每个类代表菜单上的一种饮料：

超类cost()将计算所有调料的价钱，而子类覆盖过的cost()会扩展超类的功能，把指定的饮料类型的价钱也加进来。

每个cost()方法需要计算该饮料的价钱，然后通过调用超类的cost()实现，加入调料的价钱。



Sharpen your pencil

请为下面类的cost()方法书写代码（用伪Java代码即可）。

```
public class Beverage {
   public double cost() {



   }
}
```

```
public class DarkRoast extends Beverage {

   public DarkRoast() {
      description = "Most Excellent Dark Roast";
   }

   public double cost() {



   }
}
```





Sharpen your pencil

当哪些需求或因素改变时会影响这个设计？

调料价钱的改变会使我们更改现有代码。

一旦出现新的调料，我们就需要加上新的方法，并改变超类中的cost()方法。

以后可能会开发出新饮料。对这些饮料而言（例如：冰茶），某些调料可能并不适合，但是在这个设计方式中，Tea（茶）子类仍将继承那些不适合的方法，例如：hasWhip()（加奶泡）。

万一顾客想要双倍摩卡咖啡，怎么办？

轮到你了：

这是很糟糕的！我们在第1章就得到了这个教训。



大师与门徒……

大师：我说蚱蜢呀！距离我们上次见面已经有些时日，你对于继承的冥想，可有精进？

门徒：是的，大师。尽管继承威力强大，但是我体会到它并不总是能够实现最有弹性和最好维护的设计。

大师：啊！是的，看来你已经有所长进。那么，告诉我，我的门徒，不通过继承又能如何达到复用呢？

门徒：大师，我已经了解到利用组合（composition）和委托（delegation）可以在运行时具有继承行为的效果。

大师：好，好，继续……

门徒：利用继承设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为。然而，如果能够利用组合的做法扩展对象的行为，就可以在运行时动态地进行扩展。

大师：很好，蚱蜢，你已经开始看到组合的威力了。

门徒：是的，我可以利用此技巧把多个新职责，甚至是设计超类时还没有想到的职责加在对象上。而且，可以不用修改原来的代码。

大师：利用组合维护代码，你认为效果如何？

门徒：这正是我要说的。通过动态地组合对象，可以写新的代码添加新功能，而无须修改现有代码。既然没有改变现有代码，那么引进bug或产生意外副作用的机会将大幅度减少。

大师：非常好。蚱蜢，今天的谈话就到这里。希望你能在这个主题上更深入……牢记，代码应该如同晚霞中的莲花一样地关闭（免于改变），如同晨曦中的莲花一样地开放（能够扩展）。

开放-关闭原则

此刻，蚱蜢面临最重要的设计原则之一：

设计原则

类应该对扩展开放，对修改关闭。



请进，现在“开放”中。欢迎用任何你想要的行为来扩展我们的类。如果你的需要或需求有所改变（我们知道这一定会发生的），那就来吧！动手扩展吧！



抱歉，现在是“关闭”状态。没错。我们花了许多时间得到了正确的代码，还解决了所有的bug，所以不能让你修改现有代码。我们必须关闭代码以防止被修改。如果你不喜欢，可以找经理谈。

我们的目标是允许类容易扩展，在不修改现有代码的情况下，就可搭配新的行为。如能实现这样的目标，有什么好处呢？这样的设计具有弹性可以应对改变，可以接受新的功能来应对改变的需求。

there are no Dumb Questions

问： **对扩展开放，对修改关闭？听起来很矛盾。设计如何兼顾两者？**

答： 这是一个很好的问题。乍听之下，的确感到矛盾，毕竟，越难修改的事物，就越难以扩展，不是吗？

但是，有一些聪明的OO技巧，允许系统在不修改代码的情况下，进行功能扩展。想想观察者模式（在第2章）……通过加入新的观察者，我们可以在任何时候扩展Subject（主题），而且不需向主题中添加代码。以后，你还会陆续看到更多的扩展行为的其他OO设计技巧。

问： **好吧！我了解观察者（Observ-able），但是该如何将某件东西设计成可以扩展，又禁止修改？**

答： 许多模式是长期经验的实证，可通过提供扩展的方法来保护代码免于被修改。在本章，将看到使用装饰者模式的一个好例子，完全遵循开放-关闭原则。

问： **我如何让设计的每个部分都遵循开放-关闭原则？**

答： 通常，你办不到。要让OO设计同时具备开放性和关闭性，又不修改现有的代码，需要花费许多时间和努力。一般来说，我们实在没有闲工夫把设计的每个部分都这么设计（而且，就算做得到，也可能只是一种浪费）。遵循开放-关闭原则，通常会引入新的抽象层次，增加代码的复杂度。你需要把注意力集中在设计中最有可能改变的地方，然后应用开放-关闭原则。

问： **我怎么知道，哪些地方的改变是更重要呢？**

答： 这牵涉到设计OO系统的经验，和对你工作领域的了解。多看一些其他的例子可以帮你学习如何辨别设计中的变化区。

虽然似乎有点矛盾，但是的确有一些技术可以允许在不直接修改代码的情况下对其进行扩展。

在选择需要被扩展的代码部分时要小心。每个地方都采用开放-关闭原则，是一种浪费，也没必要，还会导致代码变得复杂且难以理解。



认识装饰者模式

好了，我们已经了解利用继承无法完全解决问题，在星巴兹遇到的问题有：类数量爆炸、设计死板，以及基类加入的新功能并不适用于所有的子类。

所以，在这里要采用不一样的做法：我们要以饮料为主体，然后在运行时以调料来“装饰”（decorate）饮料。比方说，如果顾客想要摩卡和奶泡深焙咖啡，那么，要做的是：

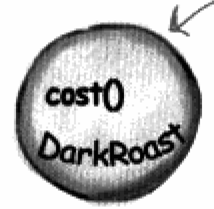
1. **拿一个深焙咖啡（DarkRoast）对象**
2. **以摩卡（Mocha）对象装饰它**
3. **以奶泡（Whip）对象装饰它**
4. **调用cost()方法，并依赖委托（delegate）将调料的价钱加上去**

好了！但是如何“装饰”一个对象，而“委托”又要如何与此搭配使用呢？给一个暗示：把装饰者对象当成“包装者”。让我们看看这是如何工作的……

以装饰者构造饮料订单

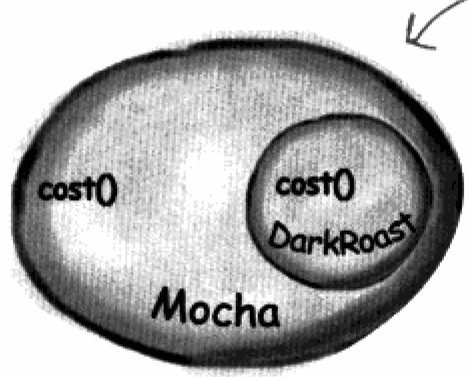
❶ 以DarkRoast对象开始

别忘了，DarkRoast继承自Beverage，且有一个用来计算饮料价钱的cost()方法。



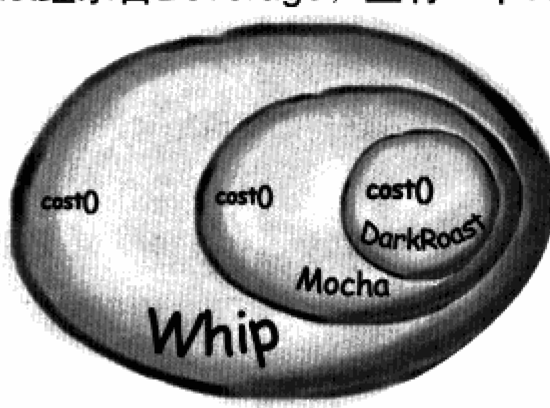
❷ 顾客想要摩卡（Mocha），所以建立一个Mocha对象，并用它将DarkRoast对象包（wrap）起来。

Mocha对象是一个装饰者，它的类型“反映”了它所装饰的对象（本例中，就是Beverage）。所谓的“反映”，指的就是两者类型一致。



所以Mocha也有一个cost()方法。通过多态，也可以把Mocha所包裹的任何Beverage当成是Beverage（因为Mocha是Beverage的子类型）。

❸ 顾客也想要奶泡（Whip），所以需要建立一个Whip装饰者，并用它将Mocha对象包起来。
别忘了，DarkRoast继承自Beverage，且有一个cost()方法，用来计算饮料价钱。



Whip是一个装饰者，所以它也反映了DarkRoast类型，并包括一个cost()方法。

所以，被Mocha和Whip包起来的DarkRoast对象仍然是一个Beverage，仍然可以具有DarkRoast的一切行为，包括调用它的cost()方法。

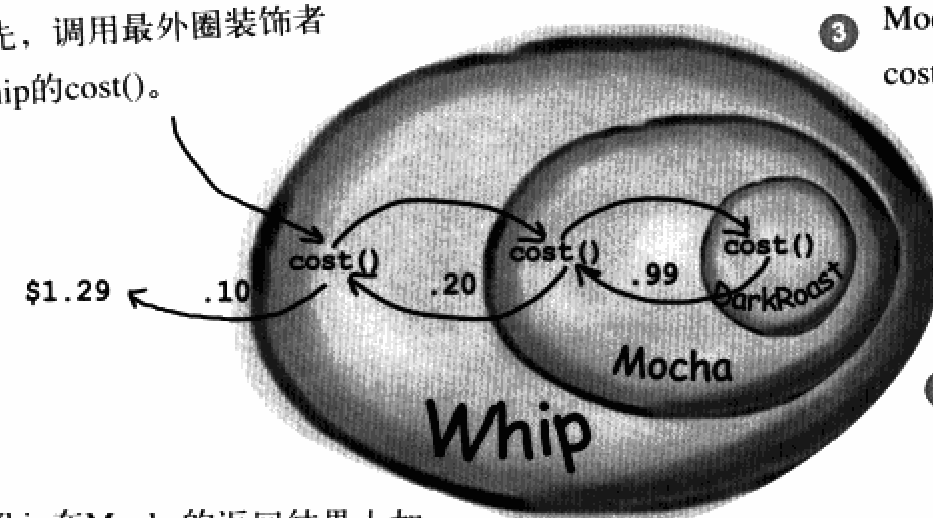
❹ 现在，该是为顾客算钱的时候了。通过调用最外圈装饰者（Whip）的cost()就可以办得到。Whip的cost()会先委托它装饰的对象（也就是Mocha）计算出价钱，然后再加上奶泡的价钱。

再过几页，你就会知道这是如何办到的。

❷ Whip调用Mocha的cost()。

❶ 首先，调用最外圈装饰者Whip的cost()。

❸ Mocha调用DarkRoast的cost()。



❹ DrakRoast返回它的价钱$0.99。

❻ Whip在Mocha的返回结果上加上自己的价钱$0.10，然后返回最后结果$1.29。

❺ Mocha在DarkRoast的结果上，加上自己的价钱$0.20，返回新的价钱$1.19。

好了，这是目前所知道的一切……

- 装饰者和被装饰对象有相同的超类型。
- 你可以用一个或多个装饰者包装一个对象。
- 既然装饰者和被装饰对象有相同的超类型，所以在任何需要原始对象（被包装的）的场合，可以用装饰过的对象代替它。
- **装饰者可以在所委托被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的。**

  关键点！
- 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰对象。

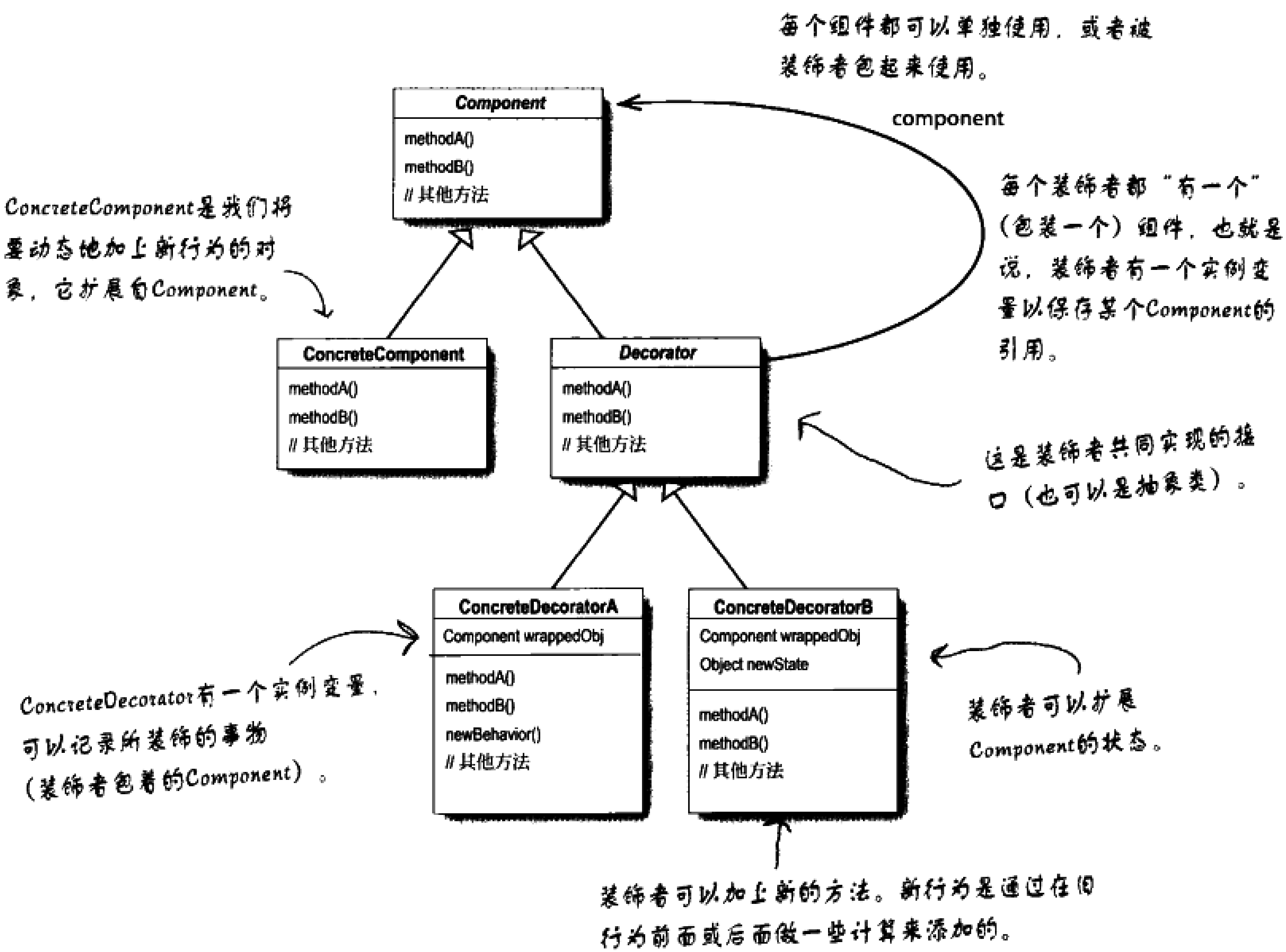
现在，就来看看装饰者模式的定义，并写一些代码，了解它到底是怎么工作的。

定义装饰者模式

让我们先来看看装饰者模式的说明：

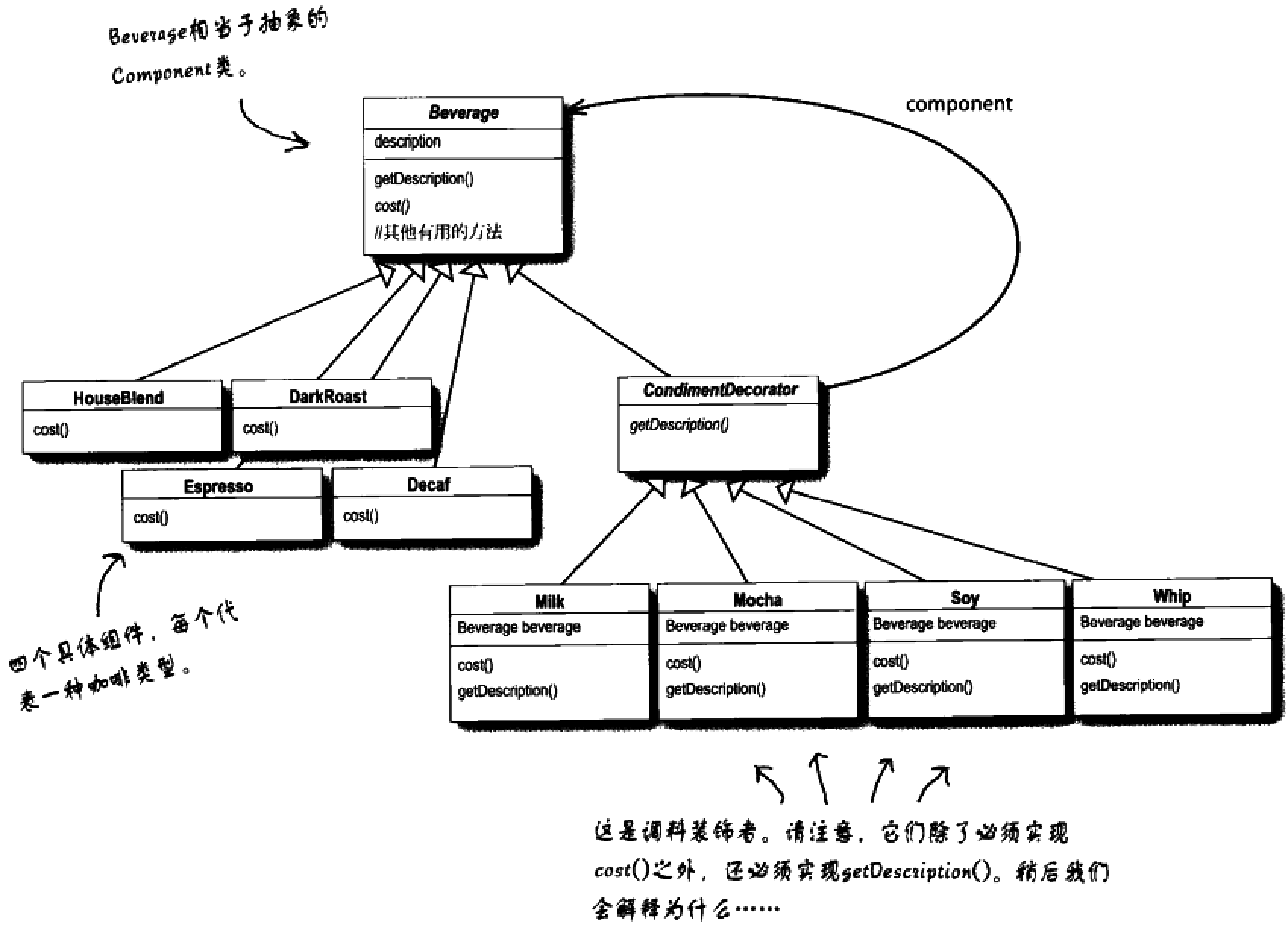
> **装饰者模式**动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

虽然这说明了装饰者模式的“角色”，但是没说明怎么在我们的实现中实际“应用”它。我们来看看类图，会有些帮助（下一页，我们会将此结构套用在饮料问题上）。



装饰我们的饮料

好吧！让星巴兹饮料也能符合此框架……

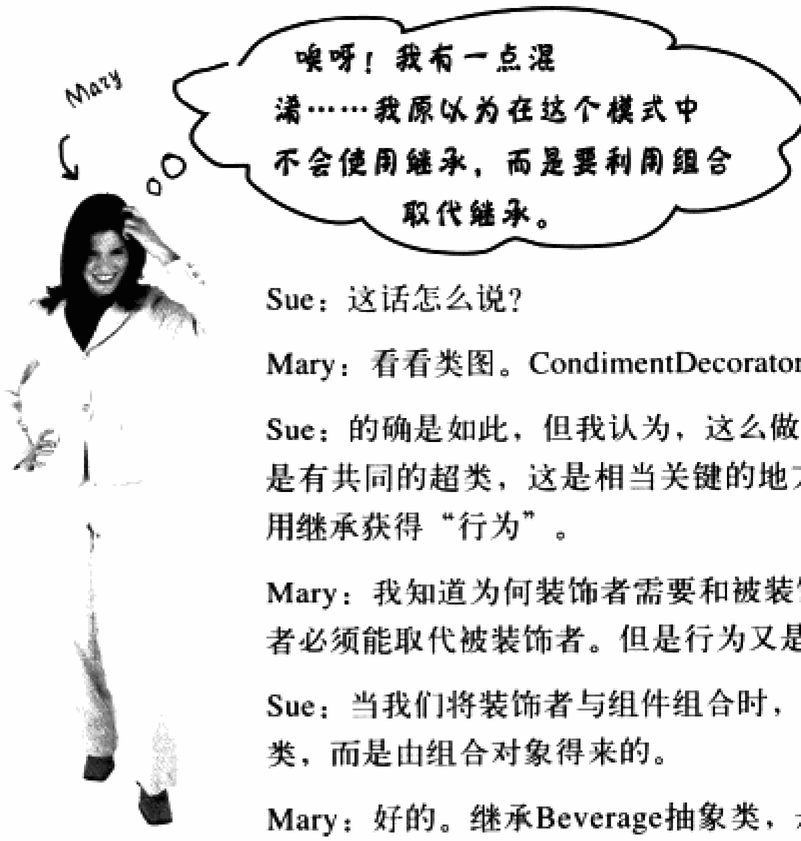


BRAIN POWER

在往下看之前，想想如何实现咖啡和调料的cost()方法。也思考一下如何实现调料的getDescription()方法。

办公室隔间对话

在继承和组合之间，观念有一些混淆。



Sue：这话怎么说？

Mary：看看类图。CondimentDecorator扩展自Beverage类，这用到了继承，不是吗？

Sue：的确是如此，但我认为，这么做的重点在于，装饰者和被装饰者必须是一样的类型，也就是有共同的超类，这是相当关键的地方。在这里，我们利用继承达到“类型匹配”，而不是利用继承获得“行为”。

Mary：我知道为何装饰者需要和被装饰者（亦即被包装的组件）有相同的“接口”，因为装饰者必须能取代被装饰者。但是行为又是从哪里来的？

Sue：当我们将装饰者与组件组合时，就是在加入新的行为。所得到的新行为，并不是继承自超类，而是由组合对象得来的。

Mary：好的。继承Beverage抽象类，是为了有正确的类型，而不是继承它的行为。行为来自装饰者和基础组件，或与其他装饰者之间的组合关系。

Sue：正是如此。

Mary：哦！我明白了。而且因为使用对象组合，可以把所有饮料和调料更有弹性地加以混和与匹配，非常方便。

Sue：是的。如果依赖继承，那么类的行为只能在编译时静态决定。换句话说，行为如果不是来自超类，就是子类覆盖后的版本。反之，利用组合，可以把装饰者混合着用……而且是在“运行时”。

Mary：而且，如我所理解的，我们可以在任何时候，实现新的装饰者增加新的行为。如果依赖继承，每当需要新行为时，还得修改现有的代码。

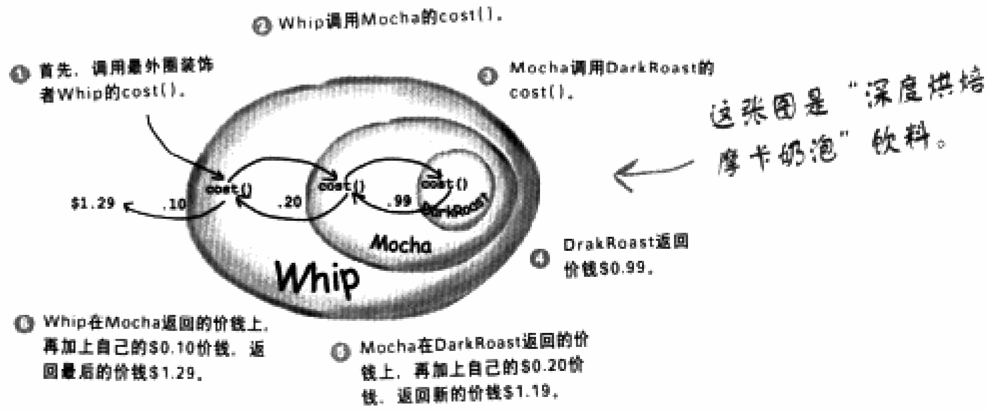
Sue：的确如此。

Mary：我还剩下一个问题，如果我们需要继承的是component类型，为什么不把Beverage类设计成一个接口，而是设计成一个抽象类呢？

Sue：关于这个嘛，还记得吗？当初我们从星巴兹拿到这个程序时，Beverage“已经”是一个抽象类了。通常装饰者模式是采用抽象类，但是在Java中可以使用接口。尽管如此，通常我们都努力避免修改现有的代码，所以，如果抽象类运作得好好的，还是别去修改它。

新咖啡师傅特训

如果有一张单子点的是：“双倍摩卡豆浆奶泡拿铁咖啡”，请使用菜单得到正确的价钱并画一个图来表达你的设计，采用和几页前一样的格式。





星巴兹咖啡

咖啡

综合	.89
深焙	.99
低咖啡因	1.05
浓缩	1.99

配料

牛奶	.10
摩卡	.20
豆浆	.15
奶泡	.10

Sharpen your pencil

把图画在这里

提示：“双倍摩卡豆浆奶泡拿铁”可以利用综合（HouseBlend）、豆浆、两小杯摩卡和奶泡来组合。

写下星巴兹的代码



该是把设计变成真正的代码的时候了！

先从Beverage类下手，这不需要改变星巴兹原始的设计。如下所示：

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage是一个抽象类，有两个方法：getDescrip-tion()及cost()。

getDescription()已经在此实现了，但是cost()必须在子类中实现。

Beverage很简单。让我们也来实现Condiment（调料）抽象类，也就是装饰者类吧：

首先，必须让Condiment Decorator能够取代Beverage，所以将Condiment Decorator扩展自 Beverage 类。

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

所有的调料装饰者都必须重新实现getDescription()方法。稍后我们会解释为什么……

写饮料的代码

现在，已经有了基类，让我们开始开始实现一些饮料吧！先从浓缩咖啡（Espresso）开始。别忘了，我们需要为具体的饮料设置描述，而且还必须实现cost()方法。

首先，让Espresso扩展自Beverage类，因为Espresso是一种饮料。

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

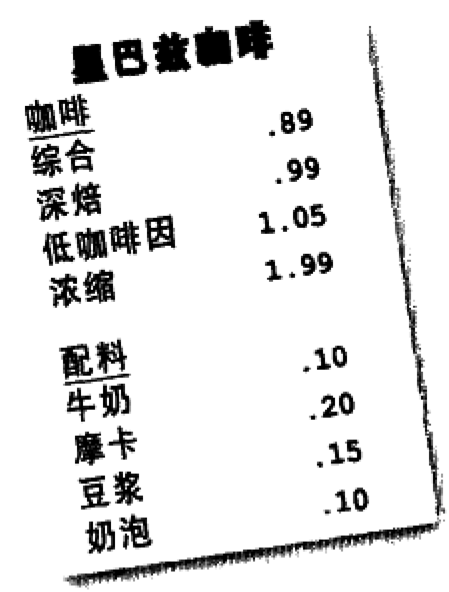
为了要设置饮料的描述，我们写了一个构造器。记住，description实例变量继承自Beverage。

最后，需要计算Espresso的价钱，现在不需要管调料的价钱，直接把Espresso的价格$1.99返回即可。

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

这是另一种饮料，做法和Espresso一样，只是把Espresso名称改为"House Blend Coffee"，并返回正确的价钱$0.89。



你可以自行建立另外两种饮料类（DarkRoast和Decaf），做法都一样。

写调料代码

如果你回头去看看装饰者模式的类图，将发现我们已经完成了抽象组件（Beverage），有了具体组件（HouseBlend），也有了抽象装饰者（CondimentDecorator）。现在，我们就来实现具体装饰者。先从摩卡下手：

摩卡是一个装饰者，所以让它扩展自CondimentDecorator。

别忘了，CondimentDecorator扩展自Beverage。

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

要让Mocha能够引用一个Beverage，做法如下：

(1)用一个实例变量记录饮料，也就是被装饰者。

(2)想办法让被装饰者（饮料）被记录到实例变量中。这里的做法是：把饮料当作构造器的参数，再由构造器将此饮料记录在实例变量中。

我们希望叙述不只是描述饮料（例如“DarkRoast”），而是完整地连调料都描述出来（例如“DarkRoast, Mocha”）。所以首先利用委托的做法，得到一个叙述，然后在其后加上附加的叙述（例如“Mocha”）。

要计算带Mocha饮料的价钱。首先把调用委托给被装饰对象，以计算价钱，然后再加上Mocha的价钱，得到最后结果。

在下一页，我们会实际实例化一个饮料对象，然后用各种调料（装饰者）包装它。但是，在这么做之前，首先……

Sharpen your pencil

写下Soy和Whip调料的代码，并完成编译。你需要它们，否则将无法进行下一页的程序。

供应咖啡

恭喜你，是时候舒服地坐下来，点一些咖啡，看看你利用装饰者模式设计出的灵活系统是多么神奇了。

这是用来下订单的一些测试代码★：

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

订一杯Espresso，不需要调料，打印出它的描述与价钱。

制造出一个DarkRoast对象。

用Mocha装饰它。

用第二个Mocha装饰它。

用Whip装饰它。

最后，再来一杯调料为豆浆、摩卡、奶泡的HouseBlend咖啡。

★ 当我们介绍到“工厂”和“生成器”设计模式时，将有更好的方式建立被装饰者对象。注意，关于“生成器模式”请参考本书附录A。

现在，来看看实验结果：

```
File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

there are no Dumb Questions

问：如果我将代码针对特定种类的具体组件（例如House-Blend），做一些特殊的事（例如，打折），我担心这样的设计是否恰当。因为一旦用装饰者包装HouseBlend，就会造成类型改变。

答：的确是这样。如果你把代码写成依赖于具体的组件类型，那么装饰者就会导致程序出问题。只有在针对抽象组件类型编程时，才不会因为装饰者而受到影响。但是，如果的确针对特定的具体组件编程，就应该重新思考你的应用架构，以及装饰者是否适合。

问：对于使用到饮料的某些客户来说，会不会容易不使用最外圈的装饰者呢？比方说，如果我有深焙咖啡，以摩卡、豆浆、奶泡来装饰，引用到豆浆而不是奶泡，代码会好写一些，这意味着订单里没有奶泡了。

答：你当然可以争辩说，使用装饰者模式，你必须管理更多的对象，所以犯下你所说的编码错误的机会会增加。但是，装饰者通常是用其他类似于工厂或生成器这样的模式创建的。一旦我们讲到这两个模式，你就会明白具体的组件及其装饰者的创建过程，它们会"封装得很好"，所以不会有这种问题。

问：装饰者知道这一连串装饰链条中其他装饰者的存在吗？比方说，我想要让getDescription()列出"Whip,Double Mocha"而不是"Mocha,Whip,Mocha"，这需要最外圈的装饰者知道有哪些装饰者牵涉其中了。

答：装饰者该做的事，就是增加行为到被包装对象上。当需要窥视装饰者链中的每一个装饰者时，这就超出他们的天赋了。但是，并不是做不到。可以写一个CondimentPrettyPrint装饰者，解析出最后的描述字符串，然后把"Mocha,Whip, Mocha"变成"Whip,Double Mocha"。如果能把getDescription()的返回值变成ArrayList类型，让每个调料名称独立开来，那么CondimentPrettyPrint方法会更容易编写。

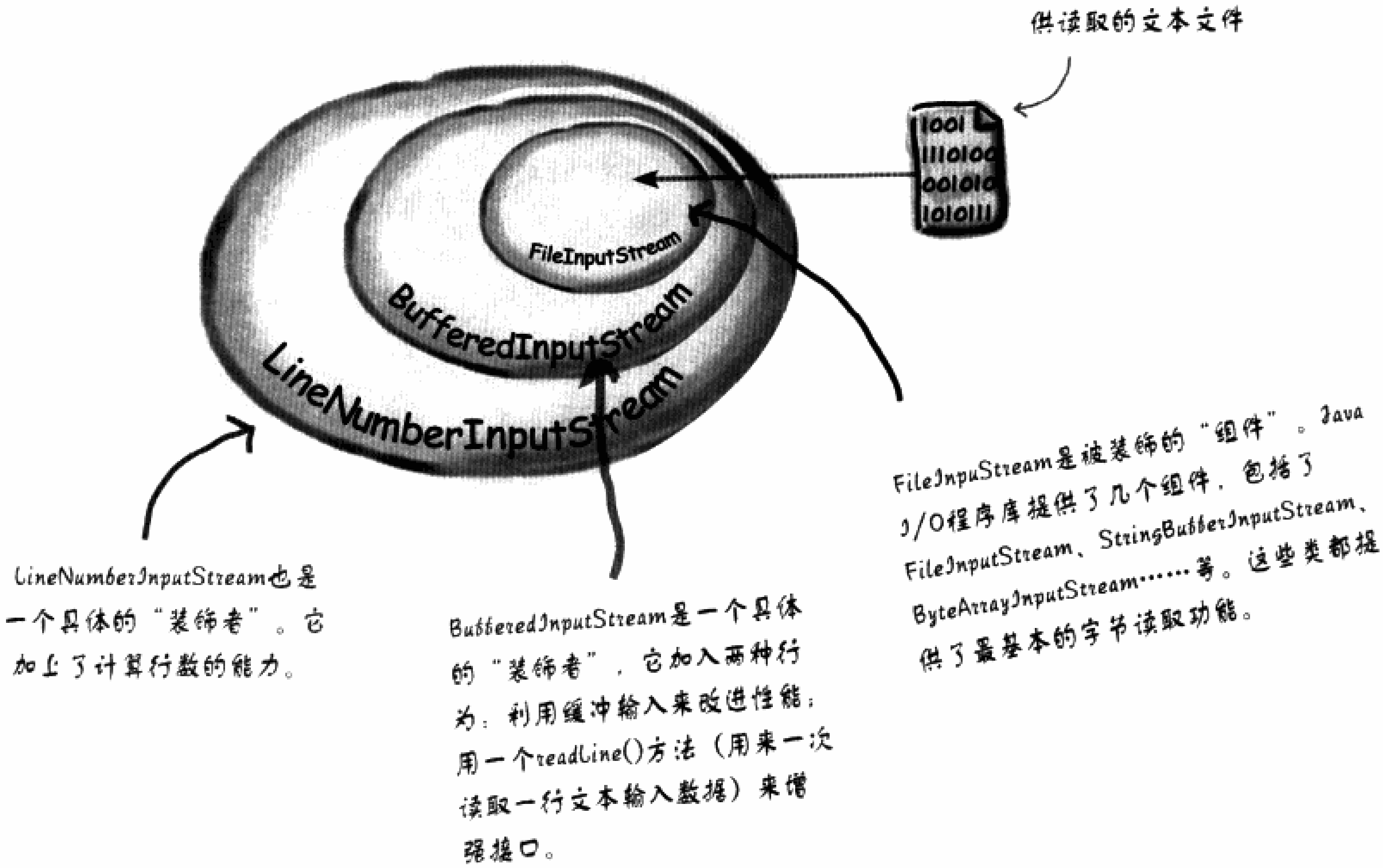
Sharpen your pencil

我们在星巴兹的朋友决定开始在菜单上加上咖啡的容量大小，供顾客可以选择小杯（tall）、中杯（grande）、大杯（venti）。星巴兹认为这是任何咖啡都必须具备的，所以在Beverage类中加上了getSize()与setSize()。他们也希望调料根据咖啡容量收费，例如：小中大杯的咖啡加上豆浆，分别加收0.10、0.15、0.20美金。

如何改变装饰者类应对这样的需求？

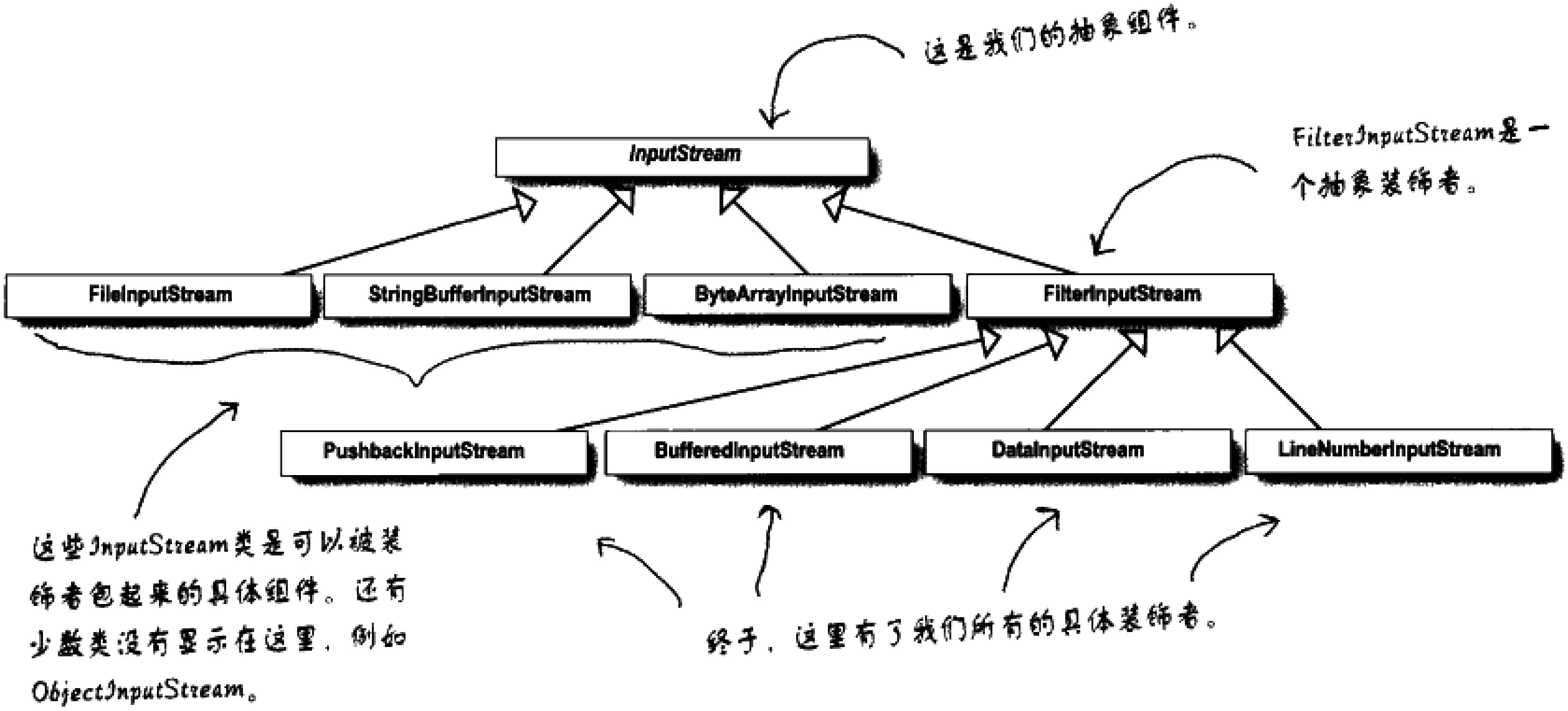
真实世界的装饰者：Java I/O

java.io包内的类太多了，简直是……“排山倒海”。你第一次（还有第二次和第三次）看到这些API发出“哇”的惊叹时，放心，你不是唯一受到惊吓的人。现在，你已经知道装饰者模式，这些I/O的相关类对你来说应该更有意义了，因为其中许多类都是装饰者。下面是一个典型的对象集合，用装饰者来将功能结合起来，以读取文件数据：



BufferedInputStream及LineNumberInputStream都扩展自FilterInputStream，而FilterInputStream是一个抽象的装饰类。

装饰java.io类



你可以发现，和星巴兹的设计相比，java.io其实没有多大的差异。我们把java.io API范围缩小，让你容易查看它的文件，并组合各种“输入”流装饰者来符合你的用途。

你会发现“输出”流的设计方式也是一样的。你可能还会发现Reader/Writer流（作为基于字符数据的输入输出）和输入流/输出流的类相当类似（虽然有一些小差异和不一致之处，但是相当雷同，所以你应该可以了解这些类）。

但是Java I/O也引出装饰者模式的一个“缺点”：利用装饰者模式，常常造成设计中有大量的小类，数量实在太多，可能会造成使用此API程序员的困扰。但是，现在你已经了解了装饰者的工作原理，以后当使用别人的大量装饰的API时，就可以很容易地辨别出他们的装饰者类是如何组织的，以方便用包装方式取得想要的行为。

编写自己的Java I/O装饰者

你已经知道装饰者模式，也看过Java I/O类图，应该已经准备好编写自己的输入装饰者了。

这个想法怎么样：编写一个装饰者，把输入流内的所有大写字符转成小写。举例：当读取“I know the Decorator Pattern therefore I RULE!”，装饰者会将它转成“i know the decorator pattern therefore i rule!”

没问题，我只要扩展FilterInputStream类，并覆盖read()方法就行了！



不要忘了导入java.io……（这里省略了）

首先，扩展FilterInputStream，这是所有InputStream的抽象装饰者。

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

现在，必须实现两个read()方法，一个针对字节，一个针对字节数组，把每个是大写字符的字节（每个代表一个字符）转成小写。

别忘了：我们在代码中没有列出package与import语句。如果想取得完整的源代码，可以到第xxxv页中列出的wickedlysmart网站URL下载。

测试你的新Java I/O装饰者

写个小程序，来测试刚写好的I/O 装饰者：

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

设置FileInputStream，先用BufferedInputStream装饰它，再用我们崭新的LowerCaseInputStream过滤器装饰它。

只用流来读取字符，一直到文件尾端。每读一个字符，就马上将它显示出来。

```
I know the Decorator Pattern therefore I RULE!
```

test.txt file

你需要做出这个文件。

运行看看：

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```



模式访谈

本周访问：
装饰者的告白

HeadFirst：欢迎装饰者模式，听说你最近情绪有点差？

装饰者：是的，我知道大家都认为我是一个有魅力的设计模式，但是，你知道吗？我也有自己的困扰，就和大家一样。

HeadFirst：愿意让我们分担一些你的困扰吗？

装饰者：当然可以。你知道我有能力为设计注入弹性，这是毋庸置疑的，但我也有“黑暗面”。有时候我会在设计中加入大量的小类，这偶尔会导致别人不容易了解我的设计方式。

HeadFirst：你能够举个例子吗？

装饰者：以Java I/O库来说，人们第一次接触到这个库时，往往无法轻易地理解它。但是如果他们能认识到这些类都是用来包装InputStream的，一切都会变得简单多了。

HeadFirst：听起来并不严重。你还是一个很好的模式，只需要一点点的教育，让大家知道怎么用，问题就解决了。

装饰者：恐怕不只这些，我还有类型问题。有些时候，人们在客户代码中依赖某种特殊类型，然后忽然导入装饰者，却又没有周详地考虑一切。现在，我的一个优点是,你通常可以透明地插入装饰者，客户程序甚至不需知道它是在和装饰者打交道。但是，如我刚刚所说的，有些代码会依赖特定的类型，而这样的代码一导入装饰者，嘭！出状况了！

HeadFirst：这个嘛,我相信每个人都必须了解到，在插入装饰者时，必须要小心谨慎。我不认为这是你的错！

装饰者：我知道，我也试着不这么想。我还有一个问题，就是采用装饰者在实例化组件时，将增加代码的复杂度。一旦使用装饰者模式，不只需要实例化组件，还要把此组件包装进装饰者中，天晓得有几个。

HeadFirst：我下周会访谈工厂（Factory）模式和生成器（Builder）模式，我听说他们对这个问题有很大的帮助。

装饰者：那倒是真的。我应该常和这些家伙聊聊。

HeadFirst：我们都认为你是一个好的模式，适合用来建立有弹性的设计，维持开放－关闭原则。你要开心一点，别负面思考。

装饰者：我尽量吧，谢谢你！



设计箱内的工具

本章已经接近尾声，你的工具箱内又多了一个新的原则和一个新的模式。

OO基础

抽象
封装
多型
继承

OO原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

为交互对象之间的松耦合设计而努力

对扩展开放，对修改关闭。

现在有了开放-关闭原则引导我们。我们会努力地设计系统，好让关闭的部分和新扩展的部分隔离。

OO模式

策略
来，
算法

观察者模式——在对象之间定义一对
多
定
并

装饰者模式——动态地将责任附加到对象上。想要扩展功能，装饰者提供有别于继承的另一种选择。

这是第一个符合开放-关闭原则的模式。真的是第一个吗？有没有其他曾经用过的遵循此原则的模式？

要点

- 继承属于扩展形式之一，但不见得是达到弹性设计的最佳方式。
- 在我们的设计中，应该允许行为可以被扩展，而无须修改现有的代码。
- 组合和委托可用于在运行时动态地加上新的行为。
- 除了继承，装饰者模式也可以让我们扩展行为。
- 装饰者模式意味着一群装饰者类，这些类用来包装具体组件。
- 装饰者类反映出被装饰的组件类型（事实上，他们具有相同的类型，都经过接口或继承实现）。
- 装饰者可以在被装饰者的行为前面与/或后面加上自己的行为，甚至将被装饰者的行为整个取代掉，而达到特定的目的。
- 你可以用无数个装饰者包装一个组件。
- 装饰者一般对组件的客户是透明的，除非客户程序依赖于组件的具体类型。
- 装饰者会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂。

习题解答

```
public class Beverage {

  //为milkCost、soyCost、mochaCost
  //和whipCost声明实例变量。
  //为milk、soy、mocha和whip
  //声明getter与setter方法。


public double cost() {

      float condimentCost = 0.0;
      if (hasMilk()) {
         condimentCost += milkCost;
      }
      if (hasSoy()) {
         condimentCost += soyCost;
      }
      if (hasMocha()) {
         condimentCost += mochaCost;
      }
      if (hasWhip()) {
         condimentCost += whipCost;
      }
      return condimentCost;
   }
}
```

```
public class DarkRoast extends Beverage {

   public DarkRoast() {
      description = "Most Excellent Dark Roast";
   }

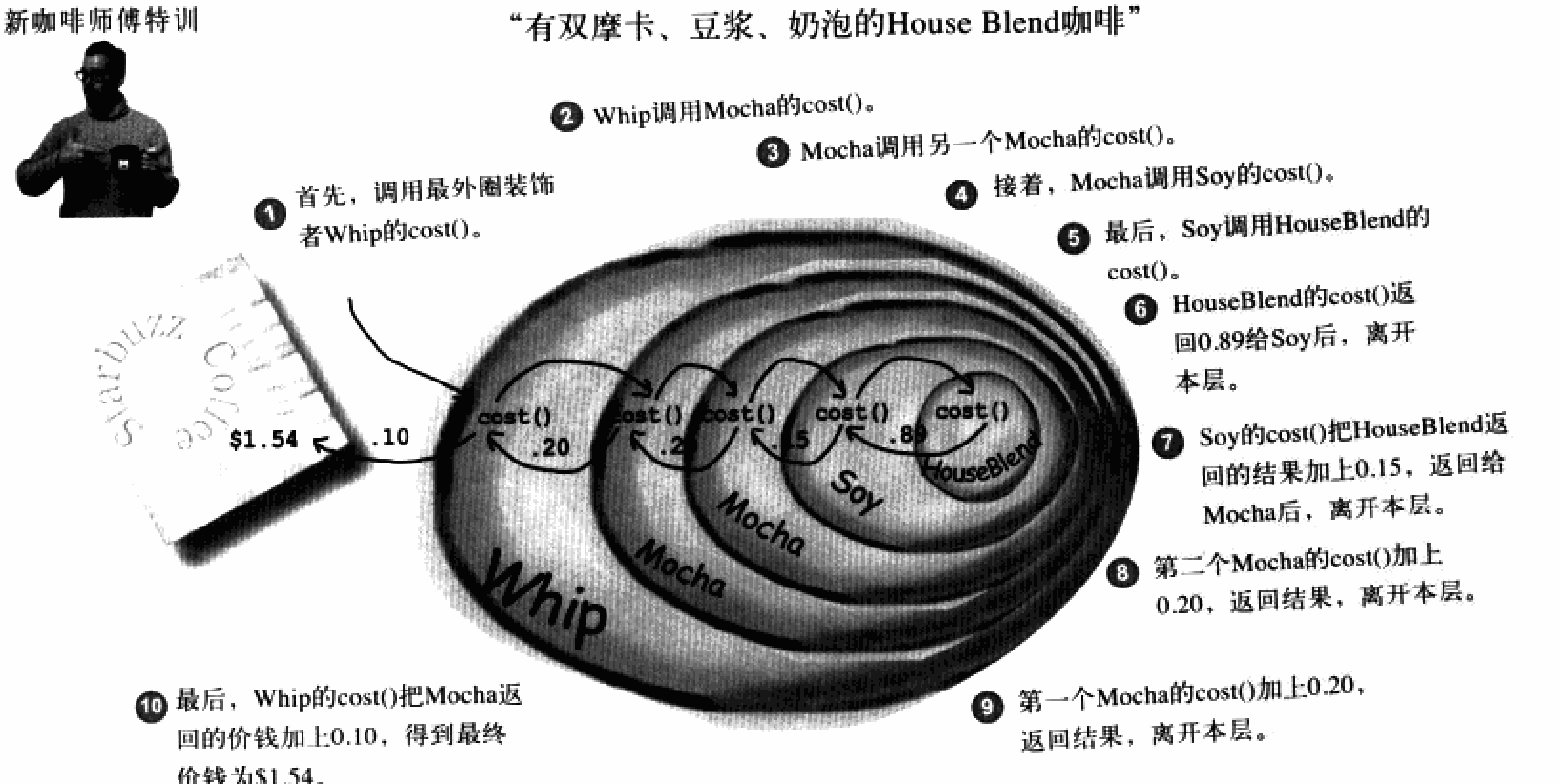
   public double cost() {

       return 1.99 + super.cost();

   }
}
```

新咖啡师傅特训

"有双摩卡、豆浆、奶泡的House Blend咖啡"



习题解答

我们在星巴兹的朋友决定开始在菜单上加上咖啡的容量大小，供顾客可以选择小杯（tall）、中杯（grande）、大杯（venti）。星巴兹认为这是任何咖啡都必须具备的，所以在Beverage类中加上了getSize()与setSize()。他们也希望调料根据咖啡容量收费，例如：小中大杯的咖啡加上豆浆，分别加收0.10、0.15、0.20美金。

如何改变装饰者类应对这样的需求？

```
public class Soy extends CondimentDecorator {
    Beverage beverage;

    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public int getSize() {
        return beverage.getSize();
    }

    public String getDescription() {
        return beverage.getDescription() +  , Soy ;
    }

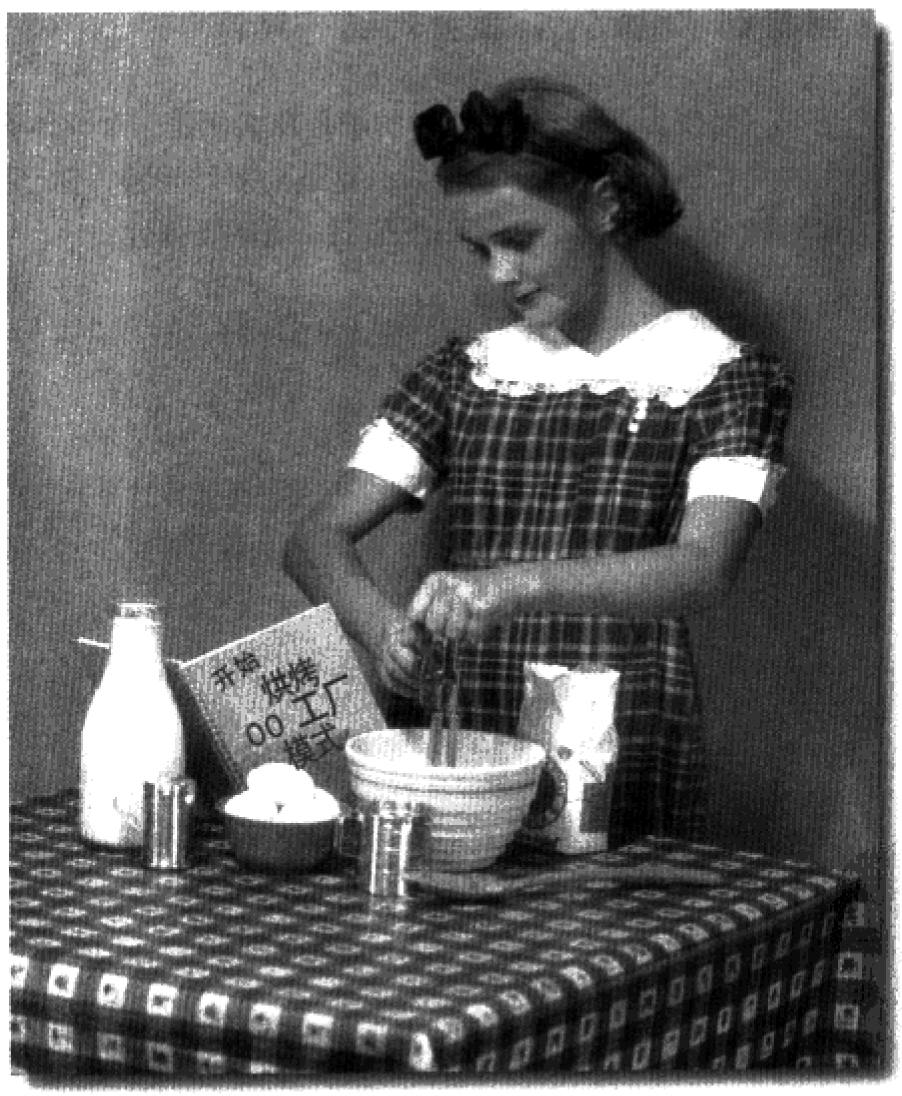
    public double cost() {
        double cost = beverage.cost();
        if (getSize() == Beverage.TALL) {
            cost += .10;
        } else if (getSize() == Beverage.GRANDE) {
            cost += .15;
        }  else if (getSize() == Beverage.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

现在要把getSize()传播到被包装的饮料。因为所有的调料装饰者都会用到这个方法，所以也应该把它移到抽象类中。

在这里取得容量大小（全都传播到具体的饮料），然后加上适当的价钱。

4 工厂模式

烘烤OO的精华



准备好开始烘烤某些松耦合的OO设计。 除了使用new操作符之外,还有更多制造对象的方法。你将了解到实例化这个活动不应该总是公开地进行，也会认识到初始化经常造成“耦合”问题。你不希望这样，对吧？读下去，你将了解工厂模式如何从复杂的依赖中帮你脱困。



当看到“new”，就会想到“具体”

是的，当使用“new”时，你的确是在实例化一个具体类，所以用的确实是实现，而不是接口。这是一个好问题，你已经知道了代码绑着具体类会导致代码更脆弱，更缺乏弹性。

```
Duck duck = new MallardDuck();
```

要使用接口让代码具有弹性

但是还是得建立具体类的实例！

当有一群相关的具体类时，通常会写出这样的代码：

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

有一大堆不同的鸭子类，但是必须等到运行时，才知道该实例化哪一个。

这里有一些要实例化的具体类，究竟实例化哪个类，要在运行时由一些条件来决定。

当看到这样的代码，一旦有变化或扩展，就必须重新打开这段代码进行检查和修改。通常这样修改过的代码将造成部分系统更难维护和更新，而且也更容易犯错。



“new”有什么不对劲？

在技术上，new没有错，毕竟这是Java的基础部分。真正的犯人是我们的老朋友“改变”，以及它是如何影响new的使用的。

针对接口编程，可以隔离掉以后系统可能发生的一大堆改变。为什么呢？如果代码是针对接口而写，那么通过多态，它可以与任何新类实现该接口。但是，当代码使用大量的具体类时，等于是自找麻烦，因为一旦加入新的具体类，就必须改变代码。也就是说，你的代码并非“对修改关闭”。想用新的具体类型来扩展代码，必须重新打开它。

所以，该怎么办？当遇到这样的问题时，就应该回到OO设计原则去寻找线索。别忘了，我们的第一个原则用来处理改变，并帮助我们“找出会变化的方面，把它们从不变的部分分离出来”。

BRAIN POWER

如何将实例化具体类的代码从应用中抽离，或者封装起来，使它们不会干扰应用的其他部分？

识别变化的方面



假设你有一个比萨店，身为对象村内最先进的比萨店主人，你的代码可能这么写：

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

为了让系统有弹性，我们很希望这是一个抽象类或接口。但如果这样，这些类或接口就无法直接实例化。

但是你需要更多比萨类型……

所以必须增加一些代码，来“决定”适合的比萨类型，然后再“制造”这个比萨：

```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek") {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

现在把比萨类型传入orderPizza()。

根据比萨的类型，我们实例化正确的具体类，然后将其赋值给pizza实例变量。请注意，这里的任何比萨都必须实现Pizza接口。

一旦我们有了一个比萨，需要做一些准备（就是擀揉面皮、加上佐料，例如芝士），然后烘烤、切片、装盒！

每个Pizza的子类型（CheesePizza、VeggiePizza等）都知道如何准备自己。

但是压力来自于增加更多的比萨类型

你发现你所有的竞争者都已经在他们的菜单中加入了一些流行风味的比萨：Clam Pizza（蛤蜊比萨）、Veggie Pizza（素食比萨）。很明显，你必须要赶上他们，所以也要把这些风味加进你的菜单中。而最近Greek Pizza（希腊比萨）卖得不好，所以你决定将它从菜单中去掉：

此代码“没有”对修改封闭。如果比萨店改变它所供应的比萨风味，就得进到这里来修改。

```
Pizza orderPizza(String type) {
      Pizza pizza;

      if (type.equals("cheese")) {
          pizza = new CheesePizza();
      } else if (type.equals("greek") {
          pizza = new GreekPizza();
      } else if (type.equals("pepperoni") {
          pizza = new PepperoniPizza();
      } else if (type.equals("clam") {
          pizza = new ClamPizza();
      } else if (type.equals("veggie") {
          pizza = new  VeggiePizza();
      }

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

（greek 分支两行被划掉）

这是变化的部分。随着时间过去，比萨菜单改变，这里就必须一改再改。

这里是我们不想改变的地方。因为比萨的准备、烘烤、包装，多年来都持续不变，所以这部分的代码不会改变，只有发生这些动作的比萨会改变。

很明显地，如果实例化“某些”具体类，将使orderPizza()出问题，而且也无法让orderPizza()对修改关闭；但是，现在我们已经知道哪些会改变，哪些不会改变，该是使用封装的时候了。

封装创建对象的代码

现在最好将创建对象移到orderPizza()之外，但怎么做呢？这个嘛，要把创建比萨的代码移到另一个对象中，由这个新对象专职创建比萨。

```
if (type.equals("cheese")) {
    pizza = new CheesePizza();
  } else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
  } else if (type.equals("clam") {
    pizza = new ClamPizza();
  } else if (type.equals("veggie") {
    pizza = new VeggiePizza();
}
```

```
Pizza orderPizza(String type) {
      Pizza pizza;

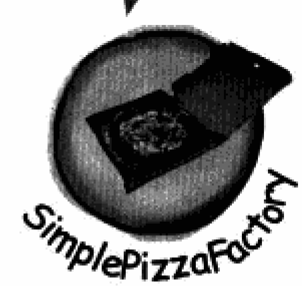


      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

首先，把创建对象的代码从orderPizza()方法中抽离。

这里该怎么写？

然后把这部分的代码搬到另一个对象中，这个新对象只管如何创建比萨。如果任何对象想要创建比萨，找它就对了。



我们称这个新对象为“工厂”。

工厂（factory）处理创建对象的细节。一旦有了SimplePizzaFactory，orderPizza()就变成此对象的客户。当需要比萨时，就叫比萨工厂做一个。那些orderPizza()方法需要知道希腊比萨或者蛤蜊比萨的日子一去不复返了。现在orderPizza()方法只关心从工厂得到了一个比萨，而这个比萨实现了Pizza接口，所以它可以调用prepare()、bake()、cut()、box()来分别进行准备、烘烤、切片、装盒。

还有一些细节有待补充，比方说，原本在orderPizza() 方法中的创建代码，现在该怎么写？现在就来为比萨店实现一个简单的比萨工厂，来研究这个问题……

建立一个简单比萨工厂

先从工厂本身开始。我们要定义一个类，为所有比萨封装创建对象的代码。代码像这样……

SimplePizzaFactory是我们的新类，它只做一件事情：帮它的客户创建比萨。

首先，在这个工厂内定义一个createPizza()方法。所有客户用这个方法来实例化新对象。

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.
equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

这是从orderPizza()方法中移过来的代码。

这个代码没什么变动，和原本orderPizza()方法中的代码一样，依然是以比萨的类型为参数。

there are no Dumb Questions

问：这么做有什么好处？似乎只是把问题搬到另一个对象罢了，问题依然存在。

答：别忘了，SimplePizzaFactory可以有许多的客户。虽然目前只看到orderPizza()方法是它的客户，然而，可能还有PizzaShopMenu（比萨店菜单）类，会利用这个工厂来取得比萨的价钱和描述。可能还有一个HomeDelivery（宅急送）类，会以与PizzaShop类不同的方式来处理比萨。总而言之，SimplePizzaFactory可以有许多的客户。

所以，把创建比萨的代码包装进一个类，当以后实现改变时，只需修改这个类即可。

别忘了，我们也正要把具体实例化的过程，从客户的代码中删除！

问：我曾看过一个类似的设计方式，把工厂定义成一个静态的方法。这有何差别？

答：利用静态方法定义一个简单的工厂，这是很常见的技巧，常被称为静态工厂。为何使用静态方法？因为不需要使用创建对象的方法来实例化对象。但请记住，这也有缺点，不能通过继承来改变创建方法的行为。

重做PizzaStore类

是时候修改我们的客户代码了，我们所要做的是仰仗工厂来为我们创建比萨，要做这样的改变：

现在我们为 PizzaStore加上一个对 SimplePizzaFactory的引用。

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    //这里是其他方法
}
```

PizzaStore的构造器，需要一个工厂作为参数。

而orderPizza()方法通过简单传入订单类型来使用工厂创建比萨。

请注意，我们把new操作符替换成工厂对象的创建方法。这里不再使用具体实例化！

BRAIN POWER

我们知道对象组合可以在运行时动态改变行为，因为我们可以更换不同的实现。在PizzaStore例子中要如何做到这点呢？有哪些工厂的实现能够被我们自由地更换？

我不知道你怎样，但我正在想的是纽约、芝加哥、加州风味的比萨工厂（别忘了还有纽黑文）。

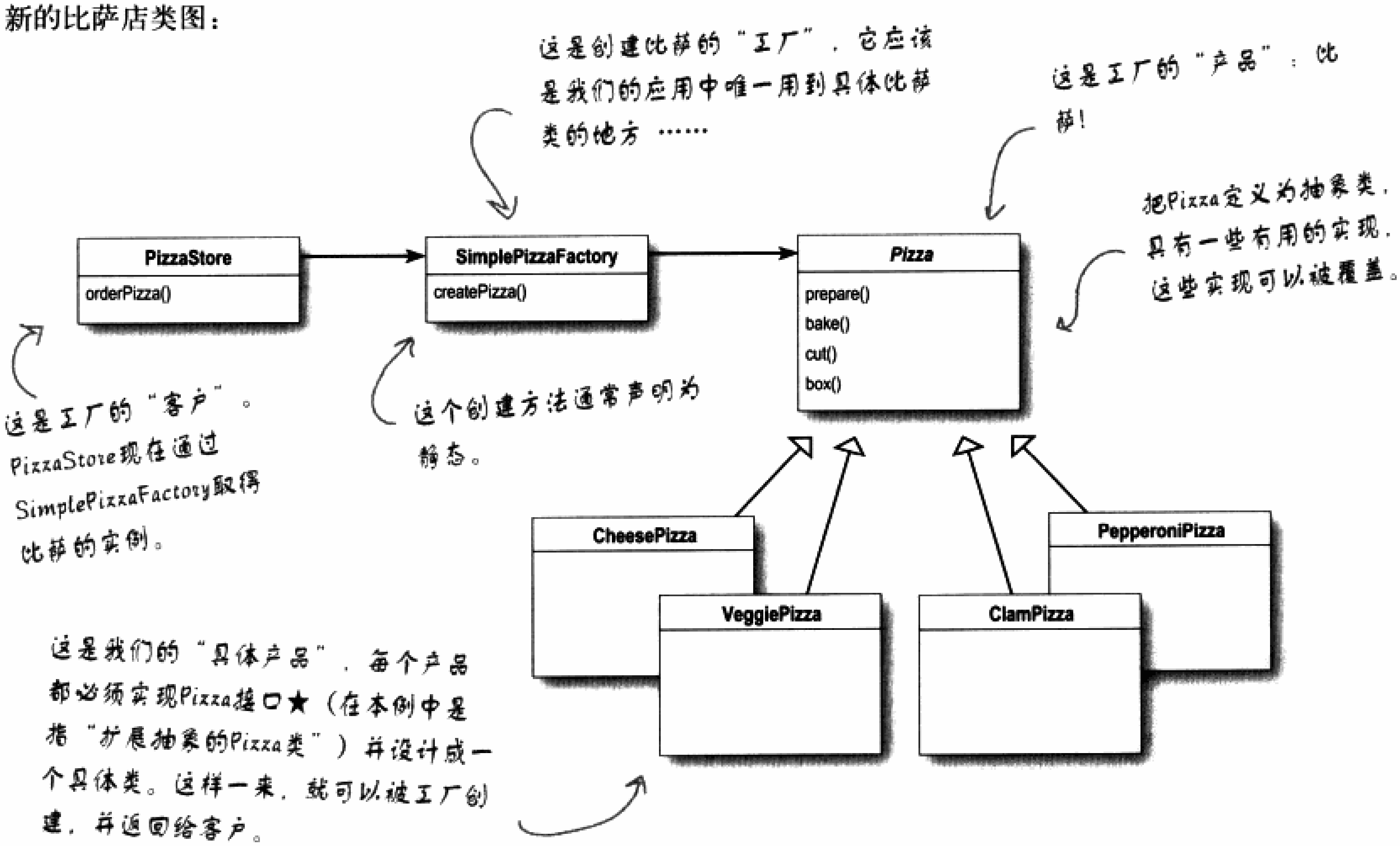
定义简单工厂



模式荣誉奖

简单工厂其实不是一个设计模式，反而比较像是一种编程习惯。但由于经常被使用，所以我们给它一个“Head First Pattern荣誉奖”。有些开发人员的确是把这个编程习惯误认为是“工厂模式”（Factory Pattern）。当你下次和另一个开发人员之间无话可说的时候，这应当是打破沉默的一个不错的话题。

不要因为简单工厂不是一个“真正的”模式，就忽略了它的用法。让我们来看看新的比萨店类图：



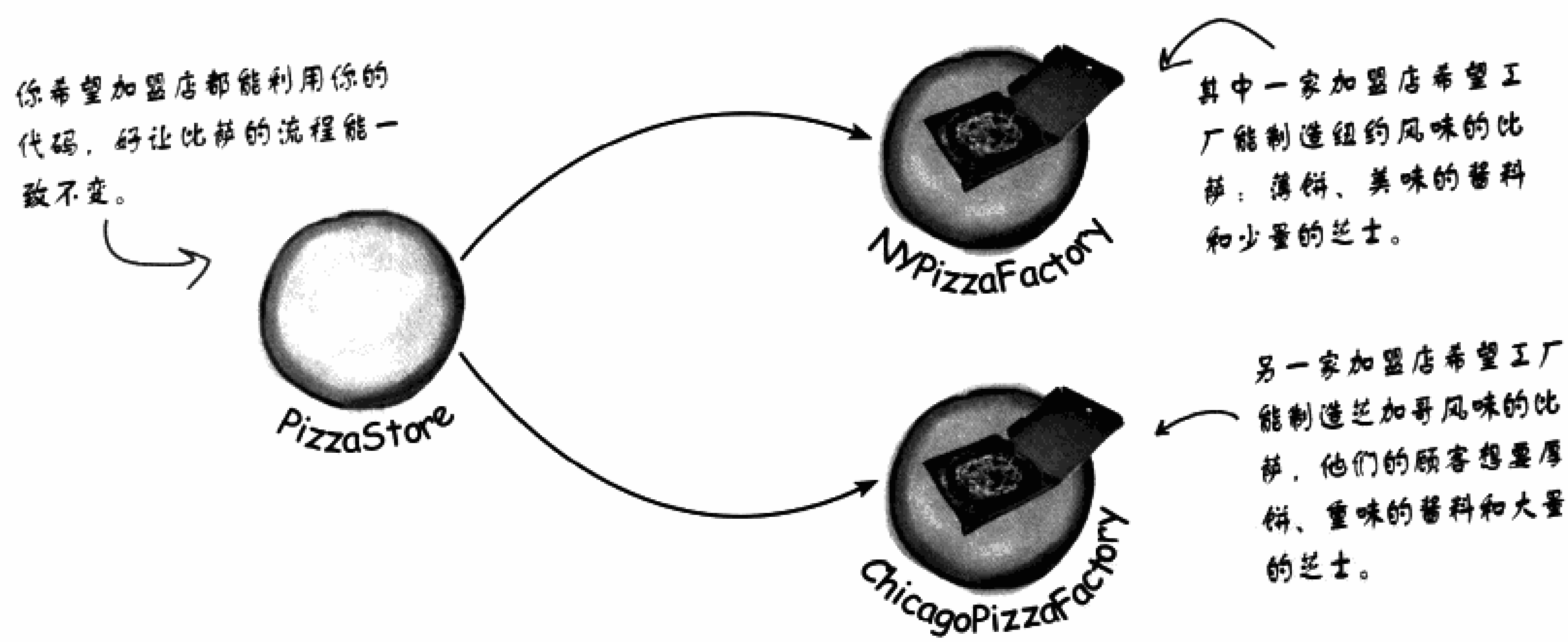
谢谢简单工厂来为我们暖身。接下来登场的是两个重量级的模式，它们都是工厂。但是别担心，未来还有更多的比萨!

★ 再提醒一次：在设计模式中，所谓的“实现一个接口”并“不一定”表示“写一个类，并利用implement关键词来实现某个Java接口”。“实现一个接口”泛指“实现某个超类型（可以是类或接口）的某个方法”。

加盟比萨店

对象村比萨店经营有成，击败了竞争者，现在大家都希望对象村比萨店能够在自家附近有加盟店。身为加盟公司经营者，你希望确保加盟店营运的质量，所以希望这些店都使用你那些经过时间考验的代码。

但是区域的差异呢？每家加盟店都可能想要提供不同风味的比萨（比方说纽约、芝加哥、加州），这受到了开店地点及该地区比萨美食家口味的影响。



我们已经有一个做法……

如果利用SimplePizzaFactory，写出三种不同的工厂，分别是NYPizzaFactory、ChicagoPizzaFactory、CaliforniaPizzaFactory，那么各地加盟店都有适合的工厂可以使用，这是一种做法。

让我们来看看会变成什么样子……

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie");
```

这里创建的工厂，是制造纽约风味的比萨。

然后建立一个比萨店，将纽约工厂的引用作为参数。

……当我们制造比萨，会得到纽约风味的比萨。

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

芝加哥比萨店也类似，先建立一个芝加哥风味工厂，并建立一个比萨店，然后结合两者。制造出的比萨，就是芝加哥风味的比萨。

但是你想要多一些质量控制……

在推广SimpleFactory时，你发现加盟店的确是采用你的工厂创建比萨，但是其他部分，却开始采用他们自创的流程：烘烤的做法有些差异、不要切片、使用其他厂商的盒子。

再想想这个问题，你真的希望能够建立一个框架，把加盟店和创建比萨捆绑在一起的同时又保持一定的弹性。

在我们稍早的SimplePizzaFactory代码之前，制作比萨的代码绑在PizzaStore里，但这么做却没有弹性。那么，该如何做才能够吃掉比萨又保有比萨呢？（译注：鱼与熊掌兼得）

我做比萨已经有好几年，所以想在比萨店的流程中，加入自己的"改良"。



一个好的加盟店，你"不需要"管他在比萨中放了什么东西。

给比萨店使用的框架

有个做法可让比萨制作活动局限于PizzaStore类，而同时又能让这些加盟店依然可以自由地制作该区域的风味。

所要做的事情，就是把createPizza()方法放回到PizzaStore中，不过要把它设置成“抽象方法”，然后为每个区域风味创建一个PizzaStore的子类。

首先，看看PizzaStore所做的改变：

现在PizzaStore是抽象的（下面解释为何如此）。

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

现在createPizza()方法从工厂对象中移回PizzaStore。

这些都没变……

现在把工厂对象移到这个方法中。

在PizzaStore里，“工厂方法”现在是抽象的。

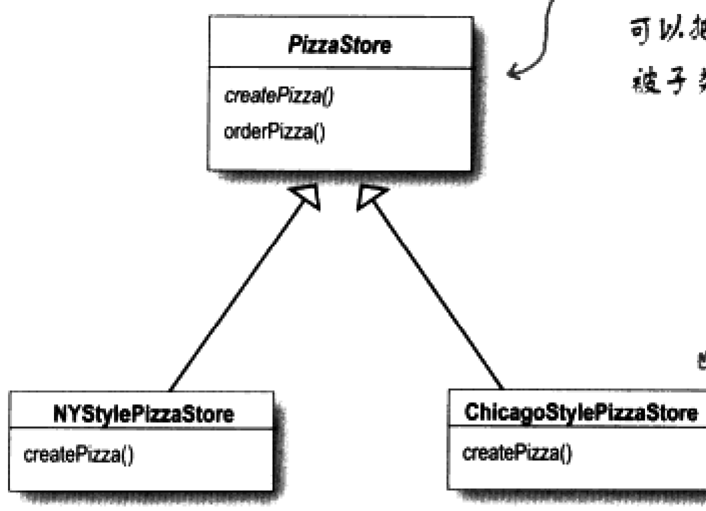
现在已经有一个PizzaStore作为超类；让每个域类型（NYPizzaStore、ChicagoPizzaStore、CaliforniaPizzaStore）都继承这个PizzaStore，每个子类各自决定如何制造比萨。让我们看看这要如何进行。

允许子类做决定

别忘了，PizzaStore已经有一个不错的订单系统，由orderPizza()方法负责处理订单，而你希望所有加盟店对于订单的处理都能够一致。

各个区域比萨店之间的差异在于他们制作比萨的风味（纽约比萨的饼薄，芝加哥比萨的饼厚等），我们现在要让createPizza()能够应对这些变化来负责创建正确种类的比萨。做法是让PizzaStore的各个子类负责定义自己的createPizza()方法。所以我们会得到一些PizzaStore具体的子类，每个子类都有自己的比萨变体，而仍然适合PizzaStore框架，并使用调试好的orderPizza()方法。

每个子类都会覆盖createPizza()方法，同时使用PizzaStore定义的orderPizza()方法。甚至可以把orderPizza()方法声明成final，以防止被子类覆盖。



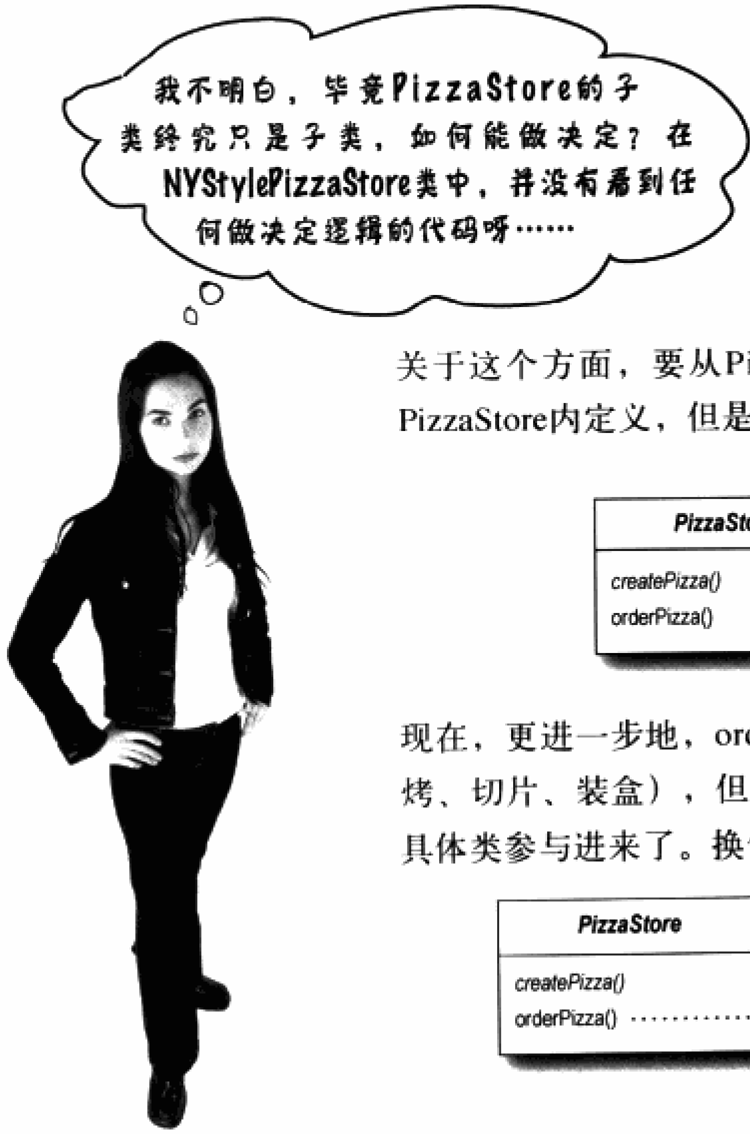
如果加盟店为顾客提供纽约风味的比萨，就使用NYStylePizzaStore，因为此类的createPizza()方法会建立纽约风味的比萨。

别忘了：PizzaStore的createPizza()是一个抽象方法，所以任何的具体比萨店子类型"必须"实现这个方法。

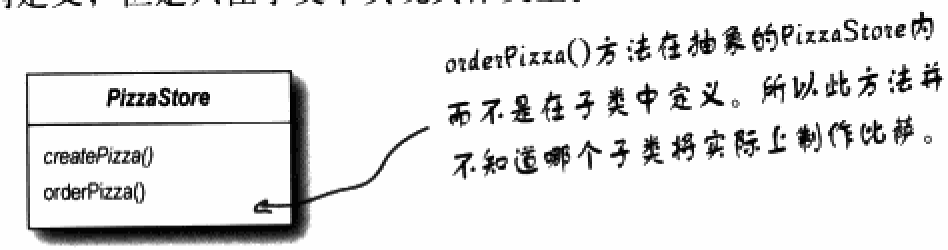
类似地，利用芝加哥子类，我们得到了带芝加哥原料的createPizza()实现。

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

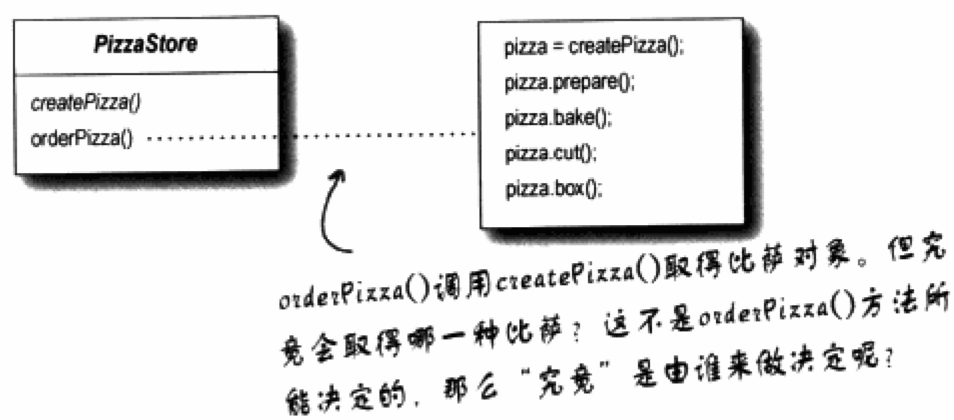
```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPiz-
za();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```



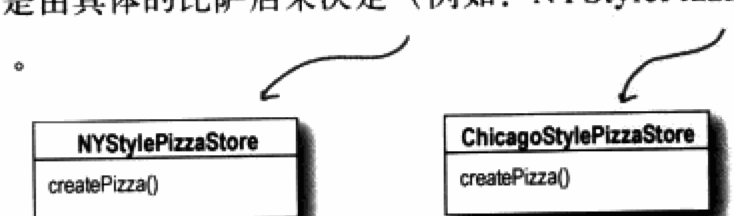
关于这个方面，要从PizzaStore的orderPizza()方法观点来看，此方法在抽象的PizzaStore内定义，但是只在子类中实现具体类型。



现在，更进一步地，orderPizza()方法对Pizza对象做了许多事情（例如：准备、烘烤、切片、装盒），但由于Pizza对象是抽象的，orderPizza()并不知道哪些实际的具体类参与进来了。换句话说，这就是解耦(decouple）！



当orderPizza()调用createPizza()时，某个比萨店子类将负责创建比萨。做哪一种比萨呢？当然是由具体的比萨店来决定（例如：NYStylePizzaStore、ChicagoStyle-PizzaStore）。



那么，子类是实时做出这样的决定吗？不是，但从orderPizza()的角度来看，如果选择在NYStylePizzaStore订购比萨，就是由这个子类（NYStylePizzaStore）决定。严格来说，并非由这个子类实际做“决定”，而是由“顾客”决定到哪一家风味的比萨店才决定了比萨的风味。

让我们开一家比萨店吧！

开加盟店有它的好处，可以从PizzaStore免费取得所有的功能。区域店只需要继承PizzaStore，然后提供createPizza()方法实现自己的比萨风味即可。这里将为加盟店处理三个比较重要的比萨风味。

这是纽约风味：

createPizza()返回一个Pizza对象，由子类全权负责该实例化哪一个具体Pizza。

NYPizzaStore扩展PizzaStore，所以拥有orderPizza()方法（以及其他的方法）。

```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

我们必须实现createPizza()方法，因为在PizzaStore里它是抽象的。

这就是创建具体类的地方。对于每一种比萨类型，都是创建纽约风味。

请注意，超类的orderPizza()方法，并不知道正在创建的比萨是哪一种，它只知道这个比萨可以被准备、被烘烤、被切片、被装盒！

一旦将这个NYPizzaStore类编译成功，不妨尝试订购一两个比萨。但在这么做之前，下一页先把芝加哥风味以及加州风味的比萨店建造完成。

Sharpen your pencil

我们已经成功地完成了NYPizzaStore，还剩下实现两个比萨店，就可以开加盟店了。请把芝加哥和加州的比萨店的实现写在这里：

声明一个工厂方法

原本是由一个对象负责所有具体类的实例化，现在通过对PizzaStore做一些小转变，变成由一群子类来负责实例化。让我们看得更仔细些：

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

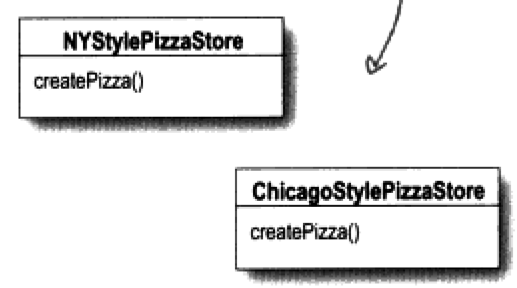
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // 其他的方法
}
```

PizzaStore的子类在createPizza()方法中，处理对象的实例化。



现在，实例化比萨的责任被移到一个"方法"中，此方法就如同是一个"工厂"

再靠近一点

工厂方法用来处理对象的创建，并将这样的行为封装在子类中。这样，客户程序中关于超类的代码就和子类对象创建代码解耦了。

工厂方法可能需要参数（也可能不需要）来指定所要的产品。

```
abstract Product factoryMethod(String type)
```

工厂方法是抽象的，所以依赖子类来处理对象的创建。

工厂方法必须返回一个产品。超类中定义的方法，通常使用到工厂方法的返回值。

工厂方法将客户（也就是超类中的代码，例如orderPizza()）和实际创建具体产品的代码分隔开来。

如何利用比萨工厂方法订购比萨



他们应该如何订购？

1. 首先，Joel和Ethan需要取得比萨店的实例。Joel需要实例化一个ChicagoPizzaStore，而Ethan需要一个NYPizzaStore。
2. 有了各自的PizzaStore，Joel和Ethan分别调用orderPizza()方法，并传入他们所喜爱的比萨类型（芝士、素食……）。
3. orderPizza()调用createPizza()创建比萨。其中NYPizzaStore实例化的是纽约风味比萨，而ChicagoPizzaStore实例化的是芝加哥风味比萨。createPizza()会将创建好的比萨当作返回值。
4. orderPizza()并不知道真正创建的是哪一种比萨，只知道这是一个比萨，能够被准备、被烘烤、被切片、被装盒，然后提供给Joel和Ethan。

看看如何根据订单生产这些比萨……

幕后花絮

1 先看看Ethan的订单：首先我们需要一个纽约比萨店：

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

建立一个NYPizza-Store的实例

2 现在有了一个店，可以下订单了：

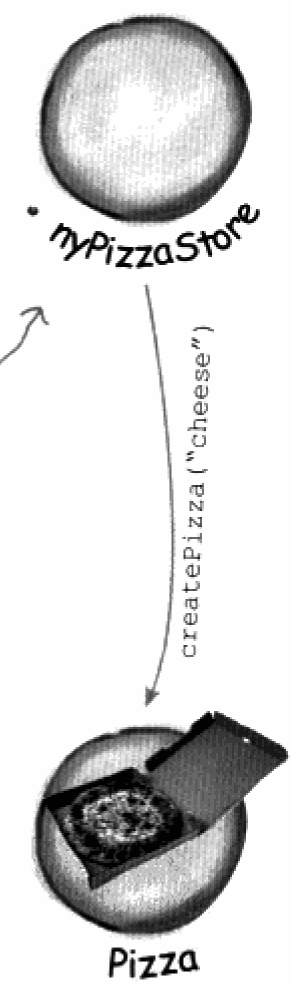
```
nyPizzaStore.orderPizza("cheese");
```

调用nyPizzaStore实例的orderPizza()方法（这个方法被定义在PizzaStore中）。

3 orderPizza()方法于是调用createPizza()方法：

```
Pizza pizza  = createPizza("cheese");
```

别忘了，工厂方法create-Pizza()是在子类中实现的。在这个例子中，它会返回纽约芝士比萨。



4 最后，比萨必须经过下列的处理才算完成 orderPizza()：

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

所制作出来的比萨一定定义了这些方法。

orderPizza()方法得到一个比萨，但不知道它实际的具体类是什么。

刚刚忽略了一件事：比萨本身！

如果没有比萨可出售，我们的比萨店开得再多也不行。现在让我们来实现比萨：



从一个抽象比萨类开始，所有的具体比萨都必须派生自这个类。

每个比萨都具有名称、面团类型、酱料类型、一套佐料。

此抽象类提供了某些默认的基本做法，用来进行烘烤、切片、装盒。

准备工作需要以特定的顺序进行，有一连串的步骤。

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (int i = 0; i < toppings.size(); i++) {
            System.out.println("   " + toppings.get(i));
        }
    }

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName() {
        return name;
    }
}
```

别忘了，这里的代码并没有提供import和package语句。如果想要完整的代码，可参考xxxv页记载的URL，到wickedlysmart网站取得。

现在我们需要一些具体子类……来定义纽约和芝加哥风味的芝士比萨，怎么样？

纽约比萨有自己的大蒜番茄酱（Marinara）和薄饼。

```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}
```

上面覆盖的是意大利reggiano高级干酪！

芝加哥比萨使用小番茄作为酱料，并使用厚饼。

```
public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

芝加哥风味的深盘比萨使用许多mozzarella（意大利白干酪）！

这个芝加哥风味比萨覆盖了cut()方法，将比萨切成正方形。

你已经等得够久了，来吃些比萨吧！

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
    }
}
```

首先建立两个不同的店。

然后用一个店帮Ethan下订单。

这个是Joel的订单。

```
File Edit Window Help YouWantMootzOnThatPizza?
%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Regiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

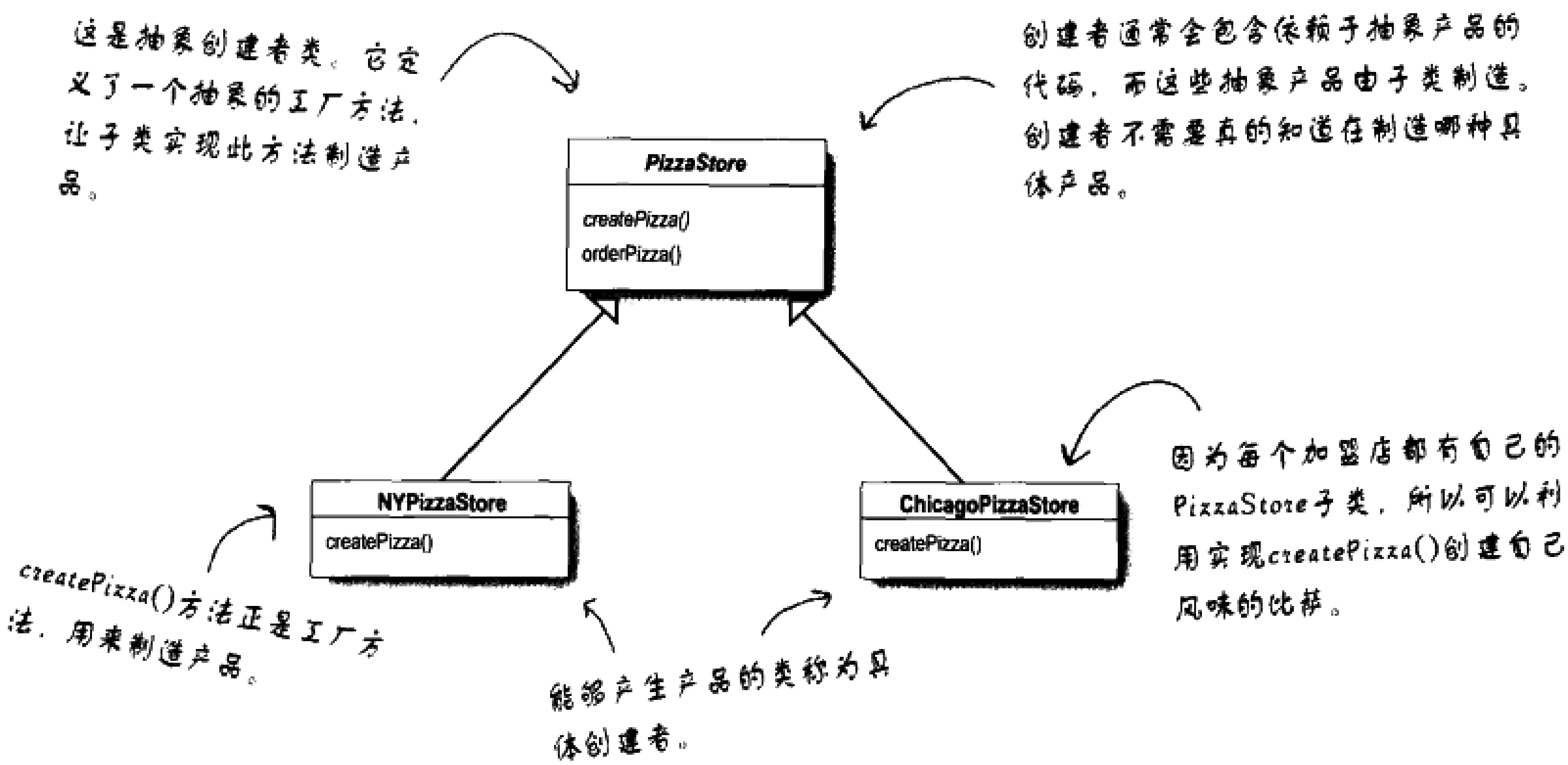
Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

两个比萨都准备好了：佐料都加上了、烘烤完成了、切片装盒了。超类从来不管细节。通过实例化正确的比萨类，子类会自行照料这一切。

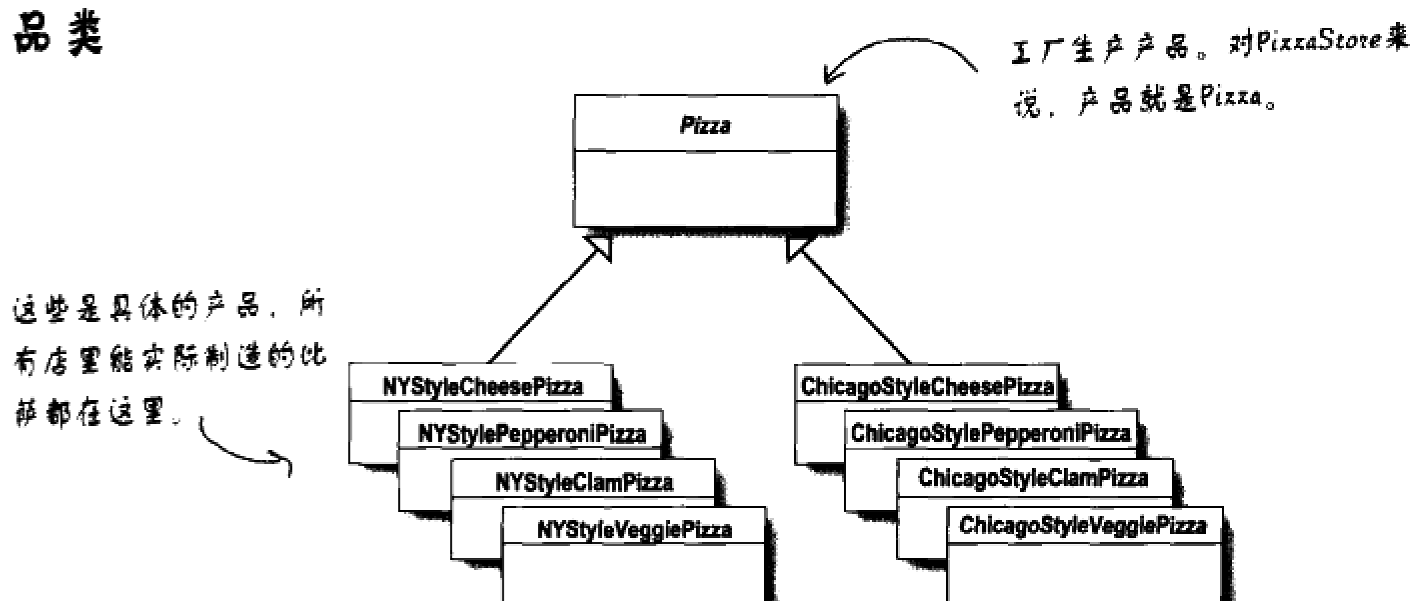
认识工厂方法模式的时刻终于到了

所有工厂模式都用来封装对象的创建。工厂方法模式（Factory Method Pattern）通过让子类决定该创建的对象是什么，来达到将对象创建的过程封装的目的。让我们来看看这些类图，以了解有哪些组成元素：

创建者（Creator）类



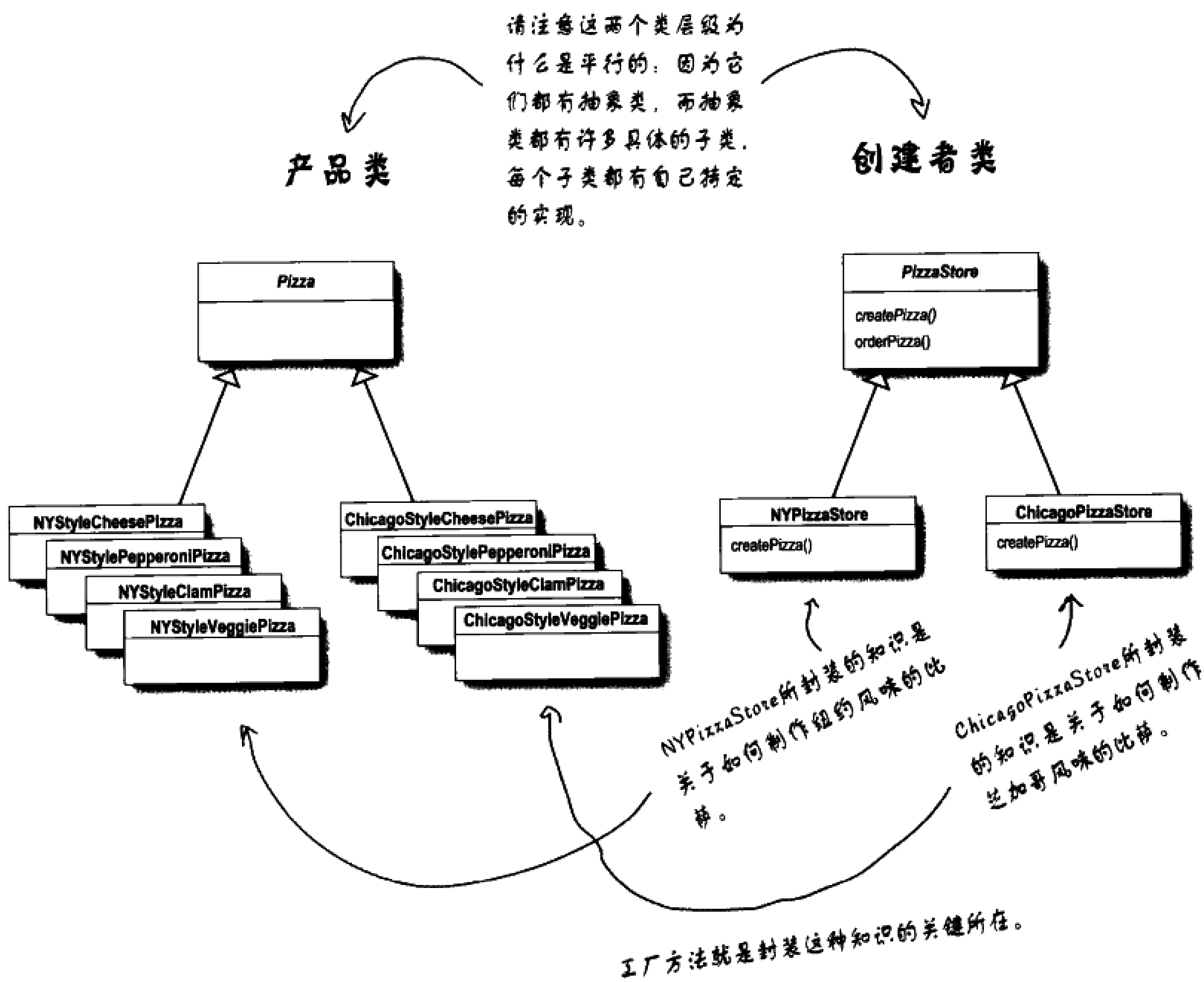
产品类



另一个观点：平行的类层级

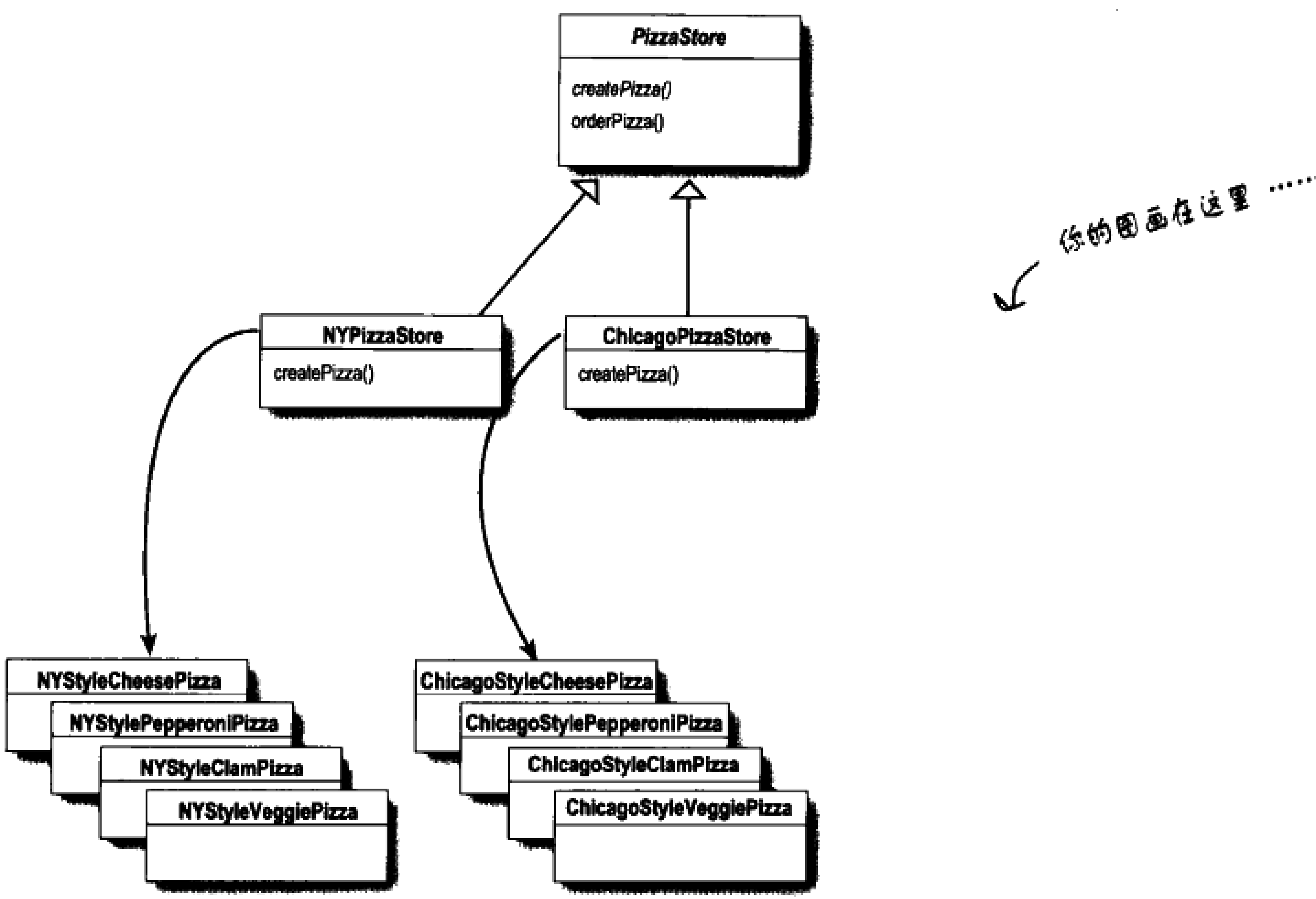
我们已经看到，将一个orderPizza()方法和一个工厂方法联合起来，就可以成为一个框架。除此之外，工厂方法将生产知识封装进各个创建者，这样的做法，也可以被视为是一个框架。

让我们来看看这两个平行的类层级，并认清它们的关系：



设计谜题

我们需要另一种比萨来符合那些疯狂加州人的需求（当然，这里的疯狂是指好的那一方面）。请绘制出另一组平行的类，把加州区域纳入PizzaStore中。



好了，发挥你的想象力，找出五个“最奇特”的东西加入到比萨中。然后你就可以准备到加州去开比萨店了！

定义工厂方法模式

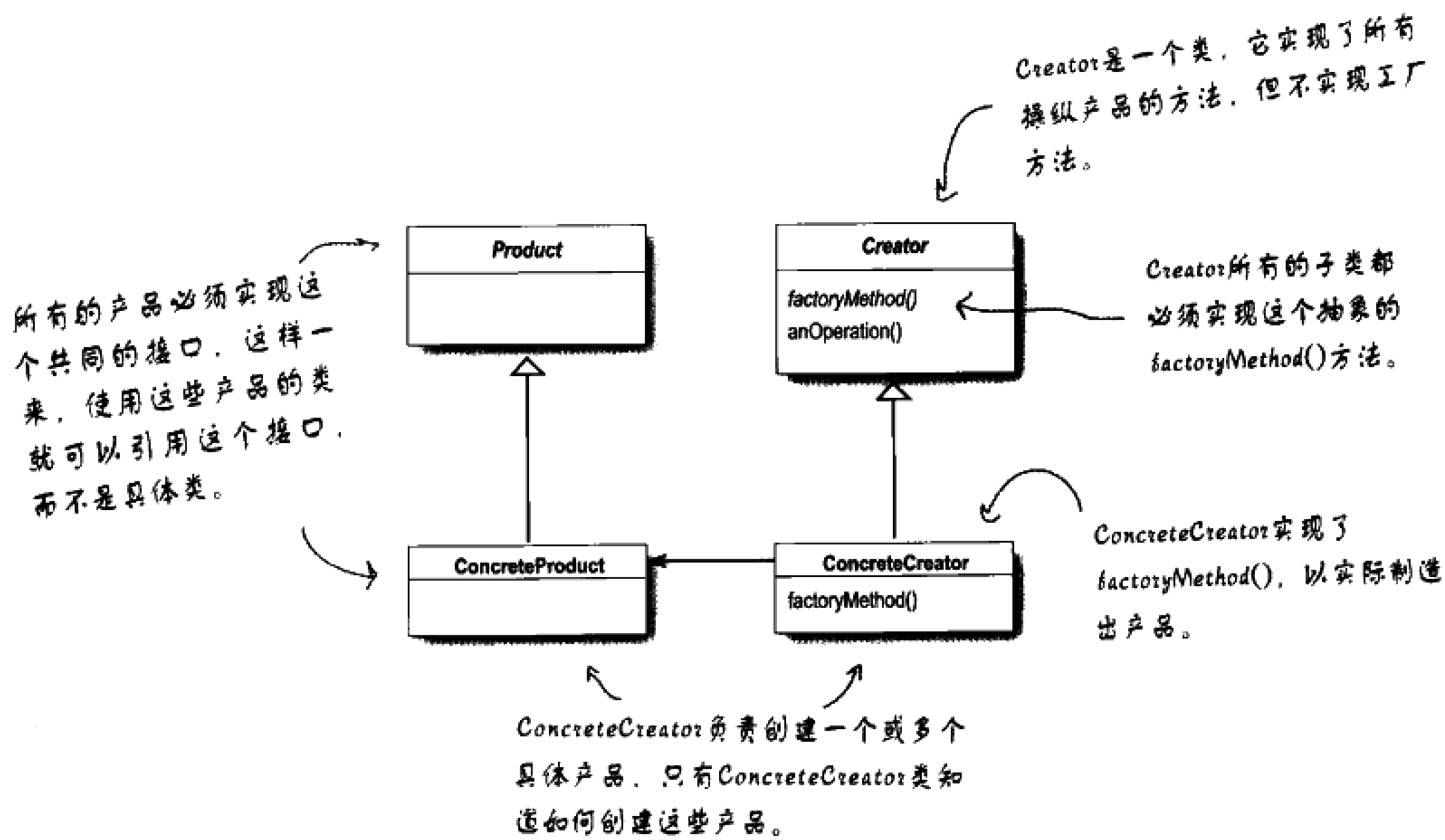
下面是工厂方法模式的正式定义:

> **工厂方法模式**定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。

工厂方法模式能够封装具体类型的实例化。看看下面的类图，抽象的Creator提供了一个创建对象的方法的接口，也称为“工厂方法”。在抽象的Creator中，任何其他实现的方法，都可能使用到这个工厂方法所制造出来的产品，但只有子类真正实现这个工厂方法并创建产品。

如同在正式定义中所说的，常常听到其他开发人员说：工厂方法让子类决定要实例化的类是哪一个。希望不要理解错误，所谓的“决定”，并不是指模式允许子类本身在运行时做决定，而是指在编写创建者类时，不需要知道实际创建的产品是哪一个。选择了使用哪个子类，自然就决定了实际创建的产品是什么。

你可以问他们，“决定”是什么意思。我敢打赌，你比他们更清楚！



there are no Dumb Questions

问：当只有一个ConcreteCreator的时候，工厂方法模式有什么优点？

答：尽管只有一个具体创建者，工厂方法模式依然很有用，因为它帮助我们将产品的“实现”从“使用”中解耦。如果增加产品或者改变产品的实现，Creator并不会受到影响（因为Creator与任何ConcreteProduct之间都不是紧耦合）。

问：如果说纽约和芝加哥的商店是利用简单工厂创建的，这样的说法是否正确？看起来倒是很像。

答：他们很类似，但用法不同。虽每个具体商店的实现看起来都很像是SimplePizza-Factory，但是别忘了，这里的具体商店是扩展自一个类，此类有一个抽象的方法createPizza()。由每个商店自行负责createPizza()方法的行为。在简单工厂中，工厂是另一个由Pizzasfore使用的对象。

问：工厂方法和创建者是否总是抽象的？

答：不，可以定义一个默认的工厂方法来产生某些具体的产品，这么一来，即使创建者没有任何子类，依然可以创建产品。

问：每个商店基于传入的类型制造出不同种类的比萨。是否所有的具体创建者都必须如此？能不能只创建一种比萨？

答：这里所采用的方式称为“参数化工厂方法”。它可以根据传入的参数创建不同的对象。然而，工厂经常只产生一种对象，不需要参数化。模式的这两种形式都是有效的。

问：利用字符串传入参数化的类型，似乎有点危险，万一把Clam（蛤蜊）英文拼错，成了Calm（平静），要求供应“CalmPizza”，怎么办？

答：说得很对，这样的情形会造成所谓的“运行时错误”。有几个其他更复杂的技巧可以避开这个麻烦，在编译时期就将参数上的错误挑出来。比方说，你可以创建代表参数类型的对象和使用静态常量或者Java 5所支持的enum。

问：对于简单工厂和工厂方法之间的差异，我依然感到困惑。他们看起来很类似，差别在于，在工厂方法中，返回比萨的类是子类。能解释一下吗？

答：子类的确看起来很像简单工厂。简单工厂把全部的事情，在一个地方都处理完了，然而工厂方法却是创建一个框架，让子类决定要如何实现。比方说，在工厂方法中，orderPizza()方法提供了一般的框架，以便创建比萨，orderPizza()方法依赖工厂方法创建具体类，并制造出实际的比萨。可通过继承PizzaStore类，决定实际制造出的比萨是什么。简单工厂的做法，可以将对象的创建封装起来，但是简单工厂不具备工厂方法的弹性，因为简单工厂不能变更正在创建的产品。



大师与门徒……

大师：蚱蜢，告诉我训练进行得如何了？

门徒：大师，我已经更进一步研究了“封装变化”。

大师：继续说……

门徒：我已经学习到，可以将创建对象的代码封装起来。实例化具体类的代码，很可能在以后经常需要变化。我学到一个称为“工厂”的技巧，可以封装实例化的行为。

大师：那么这些所谓的“工厂”究竟能带来什么好处？

门徒：有许多好处。将创建对象的代码集中在一个对象或方法中，可以避免代码中的重复，并且更方便以后的维护。这也意味着客户在实例化对象时，只会依赖于接口，而不是具体类。我在学习中发现，这可以帮助我针对接口编程，而不针对实现编程。这让代码更具有弹性，可以应对未来的扩展。

大师：很好，蚱蜢，你的OO直觉正在增强。今天对师父可有问题要问吗？

门徒：大师，我知道封装起创建对象的代码，就可以对抽象编码，将客户代码和真实的实现解耦。然而在我的工厂代码中，不可避免的，仍然必须使用具体类来实例化真正的对象。我这不是“蒙着眼睛骗自己”吗？（译注：原文pulling wool over my own eyes，作者在下面大师的回答中，将引用此句作双关语，因此如此翻译。其实，pull wool over someone's eyes原意为“骗人”）。

大师：蚱蜢呀！对象的创建是现实的，如果不创建任何对象，就无法创建任何Java程序。然而，利用这个现实的知识，可将这些创建对象的代码用栅栏围起来，就像你把所有的羊毛堆到眼前一样，一旦围起来后，就可以保护这些创建对象的代码。如果让创建对象的代码到处乱跑，那就无法收集到“羊毛”，你说是吧？

门徒：大师，我已经认识到真理。

大师：我知道你能够体会。现在请进一步调节对象的依赖。

一个很依赖的比萨店

Sharpen your pencil

假设你从未听说过OO工厂。下面是一个不使用工厂模式的比萨店版本。数一数，这个类所依赖的具体比萨对象有几种。如果又加了一种加州风味比萨到这个比萨店中，那么届时又会依赖几个对象？

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

处理所有纽约风味比萨。

处理所有芝加哥风味比萨。

可以把答案写在这里：

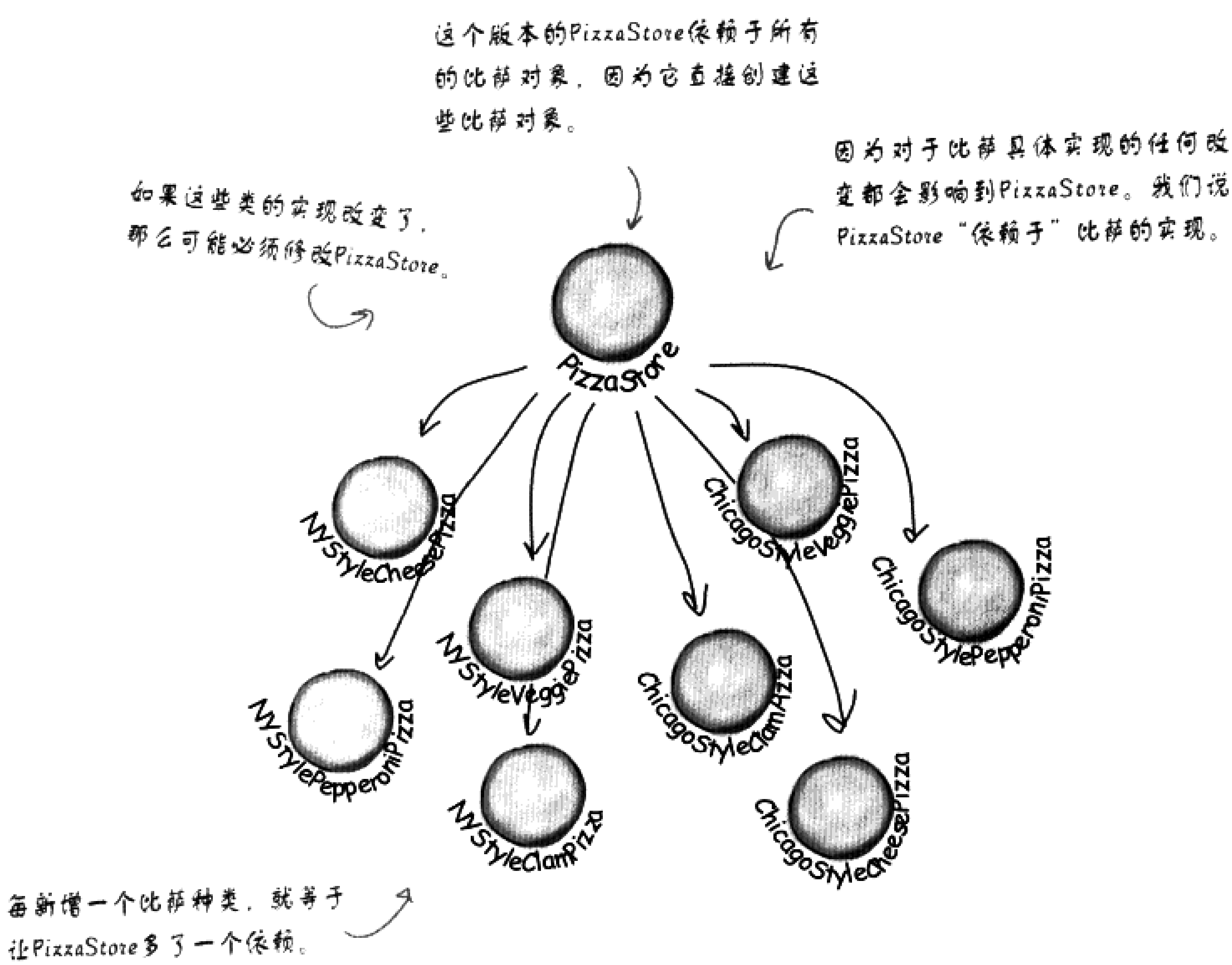
数目 ________

加上加州风味后的数目 ________

看看对象依赖

当你直接实例化一个对象时，就是在依赖它的具体类。请返回前页看看这个依赖性很高的比萨店例子，它由比萨店类来创建所有的比萨对象，而不是委托给工厂。

如果把这个版本的比萨店和它依赖的对象画成一张图，看起来是这样的：



依赖倒置原则

很清楚地，代码里减少对于具体类的依赖是件“好事”。事实上，有一个OO设计原则就正式阐明了这一点；这个原则甚至还有一个又响亮又正式的名称：“依赖倒置原则”（Dependency Inversion Principle）。

这个专有名词可以令周围的人对你刮目相看。你所获得的加薪，将比买本书所付出的钱更多，而且还会赢得其他开发人员的钦佩。

通则如下：

> **设计原则**
>
> **要依赖抽象，不要依赖具体类。**

首先，这个原则听起来很像是“针对接口编程，不针对实现编程”，不是吗？的确很相似，然而这里更强调“抽象”。这个原则说明了：不能让高层组件依赖低层组件，而且，不管高层或低层组件，“两者”都应该依赖于抽象。

所谓“高层”组件，是由其他低层组件定义其行为的类。例如，PizzaStore是个高层组件，因为它的行为是由比萨定义的：PizzaStore创建所有不同的比萨对象，准备、烘烤、切片、装盒；而比萨本身属于低层组件。

这到底是什么意思？

这个嘛，让我们再次看看前一页比萨店的图。PizzaStore是“高层组件”，而比萨实现是“低层组件”，很清楚地，PizzaStore依赖这些具体比萨类。

现在，这个原则告诉我们，应该重写代码以便于我们依赖抽象类，而不依赖具体类。对于高层及低层模块都应该如此。

但是怎么做呢？我们来想想看怎样在“非常依赖比萨店”实现中，应用这个原则……

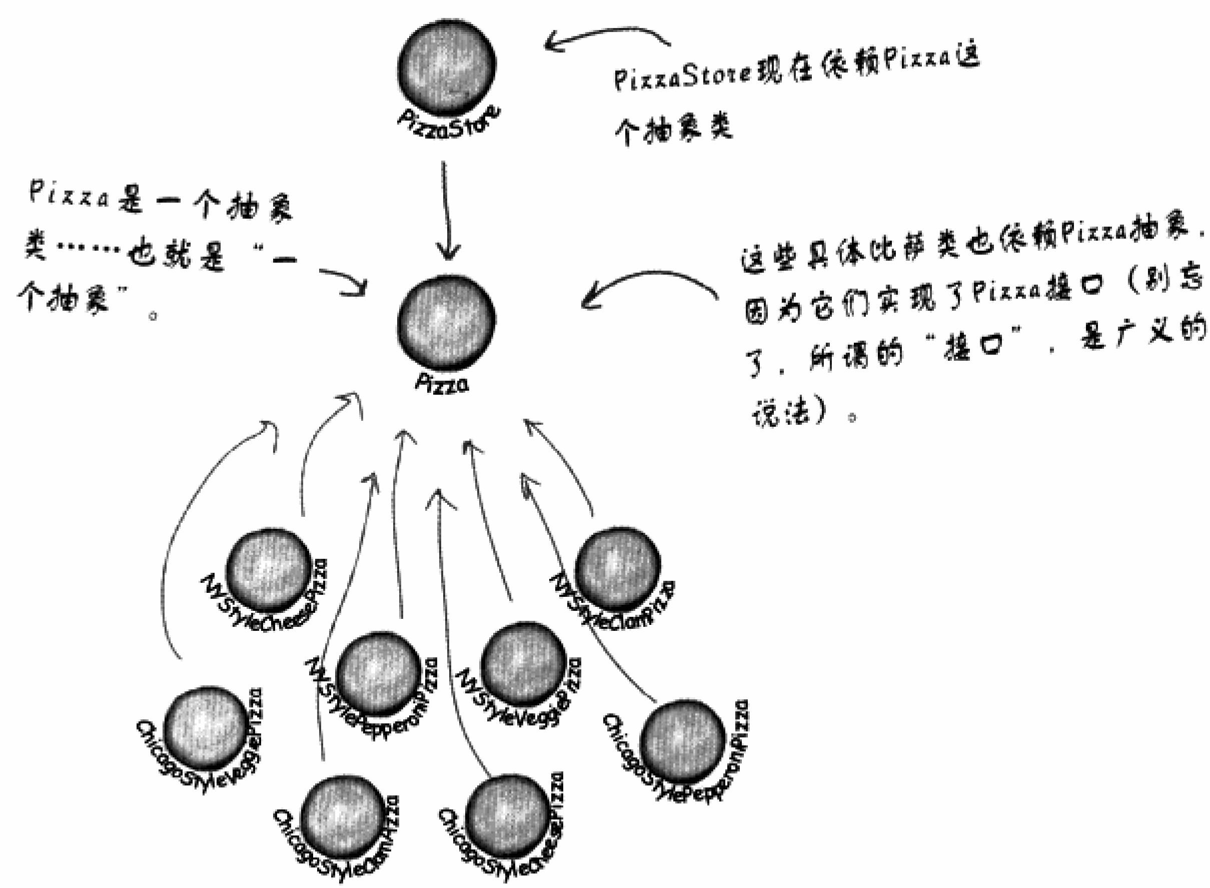
原则的应用

非常依赖比萨店的主要问题在于：它依赖每个比萨类型。因为它是在自己的orderPizza()方法中，实例化这些具体类型的。

虽然我们已经创建了一个抽象，也就是Pizza，但我们仍然在代码中，实际地创建了具体的Pizza，所以，这个抽象没什么影响力。

如何在orderPizza()方法中，将这些实例化对象的代码独立出来？我们都知道，工厂方法刚好能派上用场。

所以，应用工厂方法之后，类图看起来就像这样：



在应用工厂方法之后，你将注意到，高层组件（也就是PizzaStore）和低层组件（也就是这些比萨）都依赖了Pizza抽象。想要遵循依赖倒置原则，工厂方法并非是唯一的技巧，但却是最有威力的技巧之一。



依赖倒置原则，究竟倒置在哪里?

在依赖倒置原则中的倒置指的是和一般OO设计的思考方式完全相反。看看前一页的图，你会注意到低层组件现在竟然依赖高层的抽象。同样地，高层组件现在也依赖相同的抽象。前几页所绘制的依赖图是由上而下的，现在却倒置了，而且高层与低层模块现在都依赖这个抽象。

让我们好好地回顾一个设计过程来看看，究竟使用了这个原则之后，对设计的思考方式会被怎样地倒置……

倒置你的思考方式



好的，所以你需要实现一个比萨店，你第一件想到的事情是什么？



没错！先从顶端开始，然后往下到具体类。但是，正如你所看到的你不想让比萨店理会这些具体类，要不然比萨店将全都依赖这些具体类。现在，“倒置”你的想法……别从顶端开始，而是从比萨（Pizza）开始，然后想想看能抽象化些什么。

对了，你想要抽象化一个Pizza。好，现在回头重新思考如何设计比萨店。



很接近了，但是要这么做，必须靠一个工厂来将这些具体类取出比萨店。一旦你这么做了，各种不同的具体比萨类型就只能依赖一个抽象，而比萨店也会依赖这个抽象。我们已经倒置了一个商店依赖具体类的设计，而且也倒置了你的思考方式。

几个指导方针帮助你遵循此原则……

下面的指导方针，能帮你避免在OO设计中违反依赖倒置原则：

- 变量不可以持有具体类的引用。

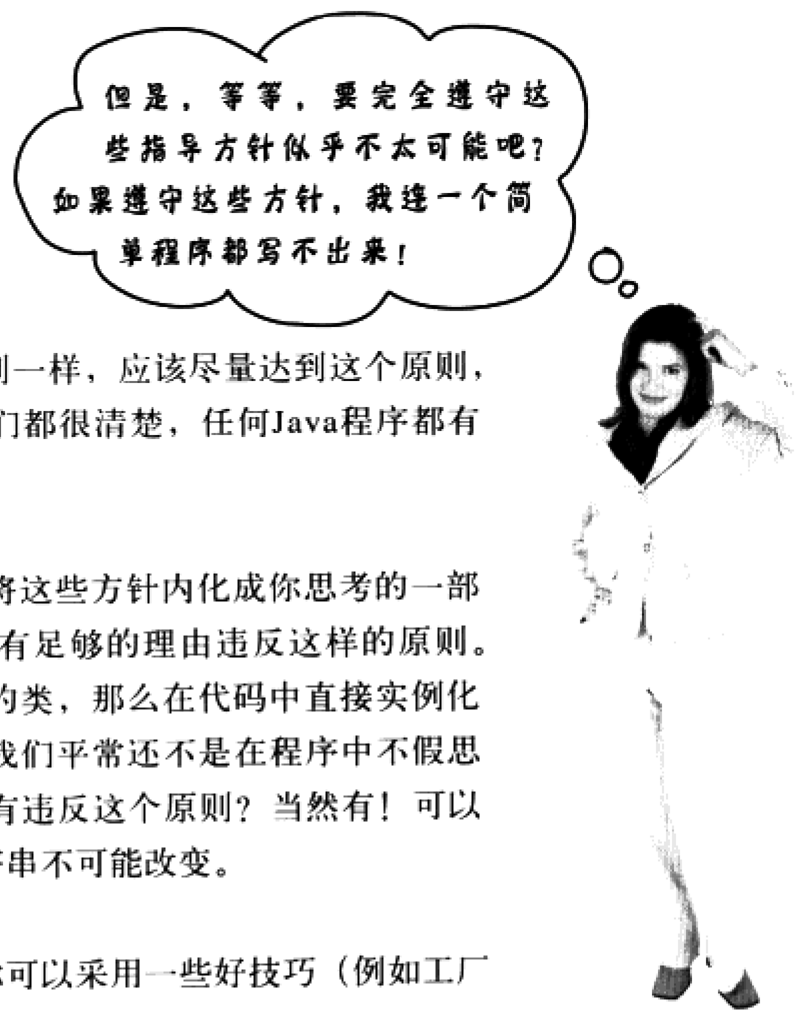
如果使用new，就会持有具体类的引用。你可以改用工厂来避开这样的做法。

- 不要让类派生自具体类。

如果派生自具体类，你就会依赖具体类。请派生自一个抽象（接口或抽象类）。

- 不要覆盖基类中已实现的方法。

如果覆盖基类已实现的方法，那么你的基类就不是一个真正适合被继承的抽象。基类中已实现的方法，应该由所有的子类共享。



你说的没错！正如同我们的许多原则一样，应该尽量达到这个原则，而不是随时都要遵循这个原则。我们都很清楚，任何Java程序都有违反这些指导方针的地方！

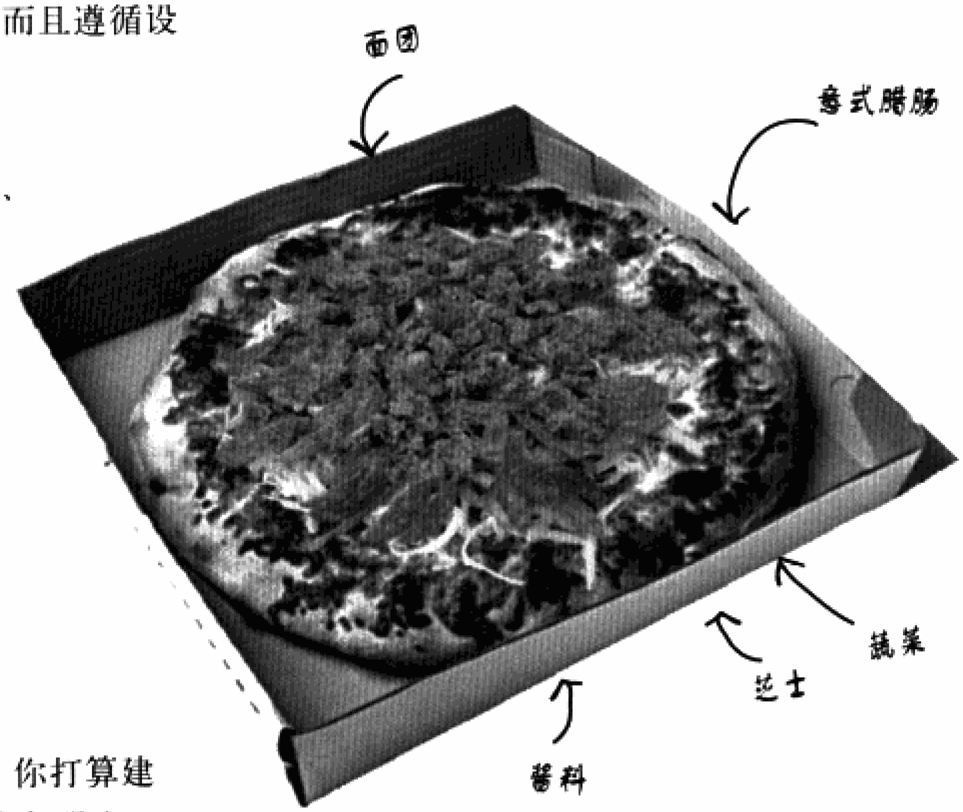
但是，如果你深入体验这些方针，将这些方针内化成你思考的一部分，那么在设计时，你将知道何时有足够的理由违反这样的原则。比方说，如果有一个不像是会改变的类，那么在代码中直接实例化具体类也就没什么大碍。想想看，我们平常还不是在程序中不假思索地就实例化字符串对象吗？就没有违反这个原则？当然有！可以这么做吗？可以！为什么？因为字符串不可能改变。

另一方面，如果有个类可能改变，你可以采用一些好技巧（例如工厂方法）来封装改变。

再回到比萨店……

比萨店的设计变得很棒：具有弹性的框架，而且遵循设计原则。

现在，对象村比萨店成功的关键在于新鲜、高质量的原料，而且通过导入新的框架，加盟店将遵循你的流程，但是有一些加盟店，使用低价原料来增加利润。你必须采取一些手段，以免长此以往毁了对象村的品牌。



确保原料的一致

要如何确保每家加盟店使用高质量的原料？你打算建造一家生产原料的工厂，并将原料运送到各家加盟店。对于这个做法，现在还剩下了一个问题：加盟店座落在不同的区域，纽约的红酱料和芝加哥的红酱料是不一样的。所以对于纽约和芝加哥，你准备了两组不同的原料。让我们看得更仔细些：

芝加哥 比萨菜单

芝士比萨
番茄酱料、意大利白干酪、Parmesan干酪、比萨草

素食比萨
番茄酱料、意大利白干酪、Parmesan干酪、茄子、菠菜、黑橄榄

蛤蜊比萨
番茄酱料、意大利白干酪、Parmesan干酪、蛤蜊

意式腊肠比萨
番茄酱料、意大利白干酪、Parmesan干酪、茄子、菠菜、黑橄榄、意式腊肠

我们有相同的产品家族（面团、意式腊肠、酱料、芝士、蔬菜、肉），但是制作方式根据区域的不同而有差异。

纽约 比萨菜单

芝士比萨
大蒜番茄酱料、Reggiano干酪、大蒜

素食比萨
大蒜番茄酱料、Reggiano干酪、蘑菇、洋葱、红椒

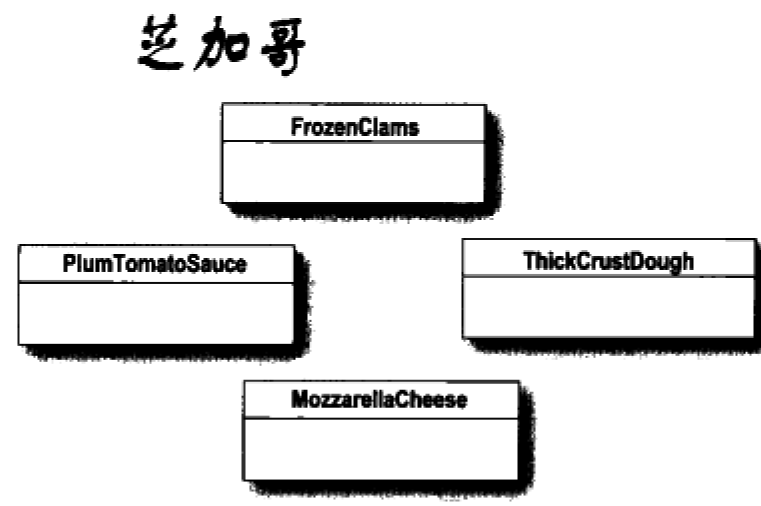
蛤蜊比萨
大蒜番茄酱料、Reggiano干酪、新鲜蛤蜊

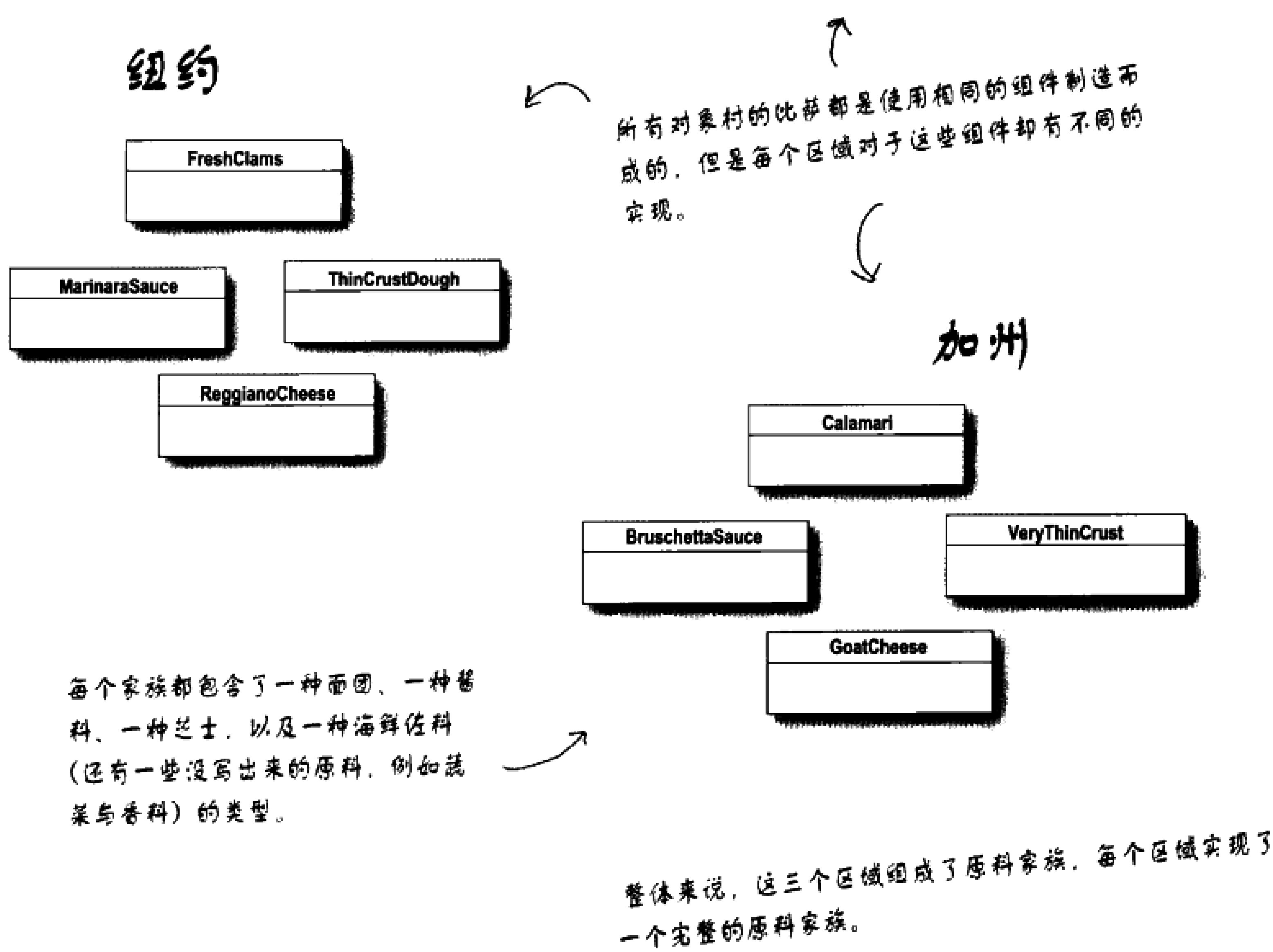
意式腊肠比萨
大蒜番茄酱料、Reggiano干酪、蘑菇、洋葱、红椒、意式腊肠

原料家族

纽约使用一组原料，而芝加哥使用另一组原料。对象村比萨是如此受欢迎，可能不久之后加州就有加盟店了，到时候又需要运送另一组区域的原料。接着呢？西雅图吗？

想要行得通，必须先清楚如何处理原料家族。





建造原料工厂

现在，我们要建造一个工厂来生产原料；这个工厂将负责创建原料家族中的每一种原料。也就是说，工厂将需要生产面团、酱料、芝士等。待会儿，你就会知道如何处理各个区域的差异了。

开始先为工厂定义一个接口，这个接口负责创建所有的原料：

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

在接口中，每个原料都有一个对应的方法创建该原料。

这里有许多新类，每个原料都是一个类。

如果每个工厂实例内都有某一种通用的"机制"需要实现，就可以把这个例子改写成抽象类……

要做的事情是：

1. 为每个区域建造一个工厂。你需要创建一个继承自PizzaIngredientFactory的子类来实现每一个创建方法。

2. 实现一组原料类供工厂使用，例如ReggianoCheese、RedPeppers、ThickCrust-Dough。这些类可以在合适的区域间共享。

3. 然后你仍然需要将这一切组织起来，将新的原料工厂整合进旧的PizzaStore代码中。

创建纽约原料工厂

好了，这是纽约原料工厂的实现。这工厂专精于大蒜番茄酱料、Reggiano干酪、新鲜蛤蜊……

具体原料工厂必须实现这个接口，纽约原料工厂也不例外。

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();
    }
}
```

对于原料家族内的每一种原料，我们都提供了纽约的版本。

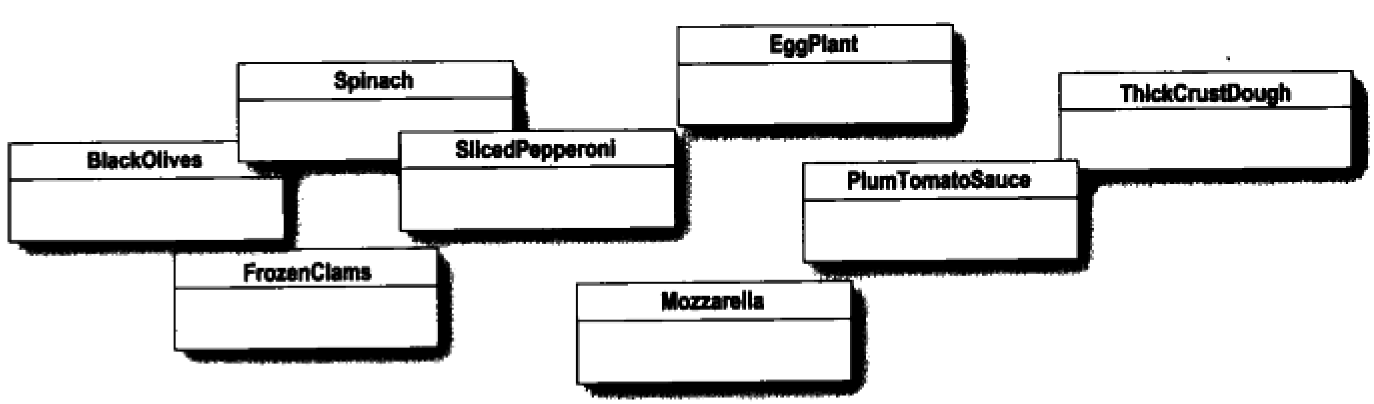
对于蔬菜，以一个蔬菜数组为返回值。在这里我们是直接把蔬菜写死。其实我们可以把它改写得更好一点，但这对于学习工厂模式并没有帮助，所以还是保持这个简单的做法就好了。

这是切片的意式腊肠，纽约和芝加哥都会用到它。在下一页，在你自己实现芝加哥工厂时，别忘了使用它。

纽约靠海，所以有新鲜的蛤蜊。芝加哥就必须使用冷冻的蛤蜊。

Sharpen your pencil

写下ChicagoPizzaIngredientFactory的代码。你可以参考下面的类，写出你的实现：



重做比萨……

工厂已经一切就绪，准备生产高质量原料了；现在我们只需要重做比萨，好让它们只使用工厂生产出来的原料。我们先从抽象的Pizza类开始：

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // 这里是打印比萨的代码
    }
}
```

每个比萨都持有一组在准备时会用到的原料。

现在把prepare()方法声明成抽象。在这个方法中，我们需要收集比萨所需的原料，而这些原料当然是来自原料工厂了。

其他的方法维持不变，只有prepare()需要改变。

继续重做比萨……

现在已经有了一个抽象比萨，可以开始创建纽约和芝加哥风味的比萨了。从今以后，加盟店必需直接从工厂取得原料，那些偷工减料的日子宣告结束了！

我们曾经写过工厂方法的代码，有NYCheesePizza和ChicagoCheesePizza类。比较一下这两个类，唯一的差别在于使用区域性的原料，至于比萨的做法都一样（面团+酱料+芝士），其他的比萨（蔬菜、蛤蜊等）也是如此。它们都依循着相同的准备步骤，只是使用不同的原料。

所以，其实我们不需要设计两个不同的类来处理不同风味的比萨，让原料工厂处理这种区域差异就可以了。下面是CheesePizza：

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

要制作比萨，需要工厂提供原料。所以每个比萨类都需要从构造器参数中得到一个工厂，并把这个工厂存储在一个实例变量中。

神奇的事情发生在这里！

prepare()方法一步一步地创建芝士比萨，每当需要原料时，就跟工厂要。

再靠近一点

Pizza的代码利用相关的工厂生产原料。所生产的原料依赖所使用的工厂，Pizza类根本不关心这些原料，它只知道如何制作比萨。现在，Pizza和区域原料之间被解耦，无论原料工厂是在洛基山脉还是在西北沿岸地区，Pizza类都可以轻易地复用，完全没有问题。

```
sauce = ingredientFactory.createSauce();
```

把Pizza的实例变量设置为此比萨所使用的某种酱料。

这是原料工厂，Pizza不在乎使用什么工厂，只要是原料工厂就行了。

createSource()方法会返回这个区域所使用的酱料。如果这是一个纽约原料工厂，我们将取得大蒜番茄酱料。

也来看看蛤蜊比萨：

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

蛤蜊比萨也需要原料工厂。

要做出蛤蜊比萨，prepare()方法就必须从本地工厂中取得正确的原料。

如果是纽约工厂，就会使用新鲜的蛤蜊；如果是芝加哥工厂，就是冷冻的蛤蜊。

再回到比萨店

我们几乎完工了，只需再到加盟店短暂巡视一下，确认他们使用了正确的比萨。也需要让他们能和本地的原料工厂搭上线：

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

纽约店会用到纽约比萨原料工厂，由该原料工厂负责生产所有纽约风味比萨所需的原料。

把工厂传递给每一个比萨，以便比萨能从工厂中取得原料。

看看前一页，确定你了解了比萨和工厂之间的关系是如何运作的。

对于每一种比萨，我们实例化一个新的比萨，并传递该种比萨所需的工厂，以便比萨取得它的原料。

BRAIN POWER

比较一下这个版本的createPizza()和之前的工厂方法实现有何异同。

我们做了些什么？

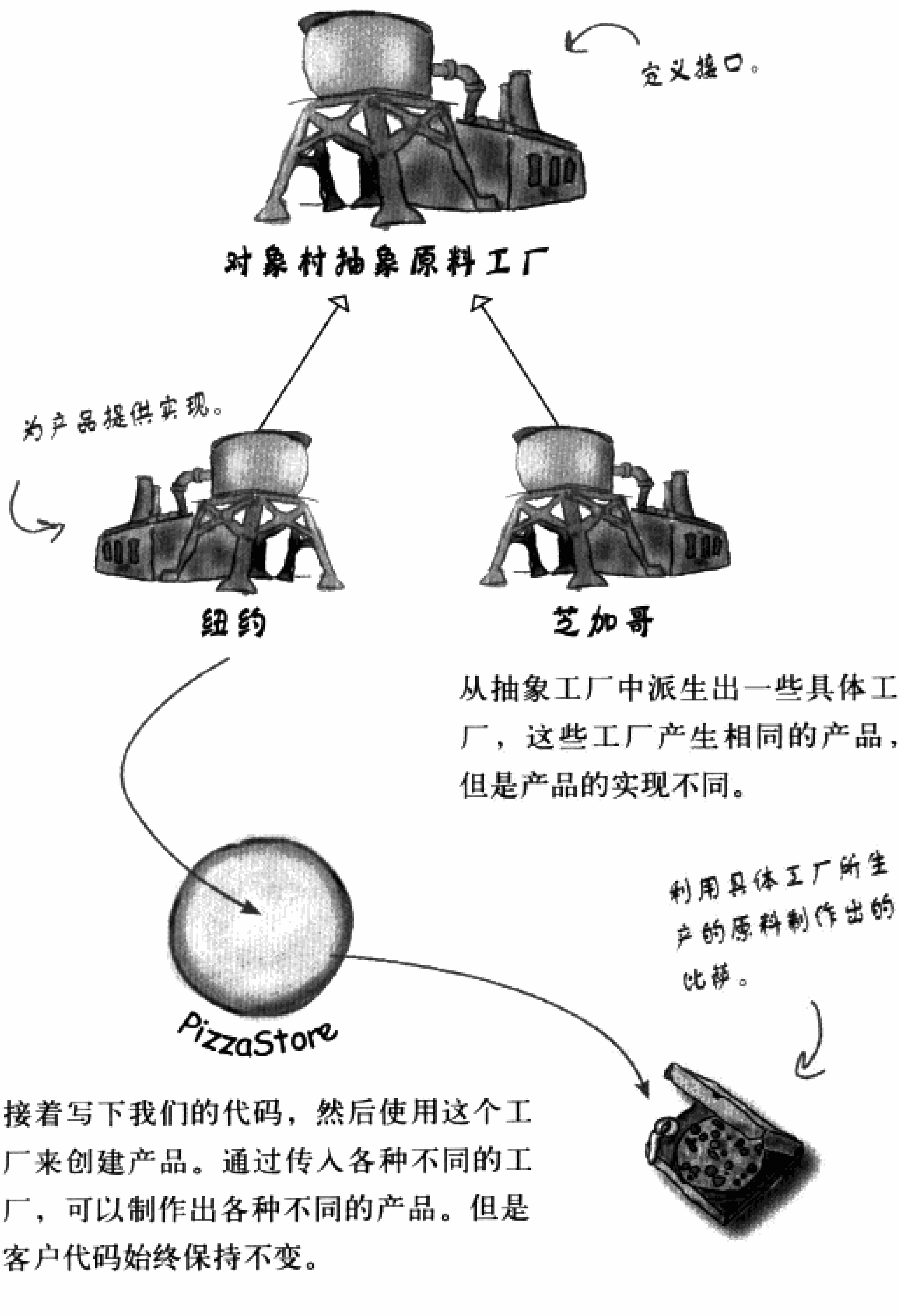
一连串的代码改变；我们到底做了些什么？

我们引入新类型的工厂，也就是所谓的抽象工厂，来创建比萨原料家族。

通过抽象工厂所提供的接口，可以创建产品的家族，利用这个接口书写代码，我们的代码将从实际工厂解耦，以便在不同上下文中实现各式各样的工厂，制造出各种不同的产品。例如：不同的区域、不同的操作系统、不同的外观及操作。

因为代码从实际的产品中解耦了，所以我们可以替换不同的工厂来取得不同的行为（例如：取得大蒜番茄酱料，而不是取得番茄酱料）。

抽象工厂为产品家族提供接口。是什么家族？在我们的例子中，制作比萨所需要的一切东西，例如：面团、酱料、芝士、肉和蔬菜。



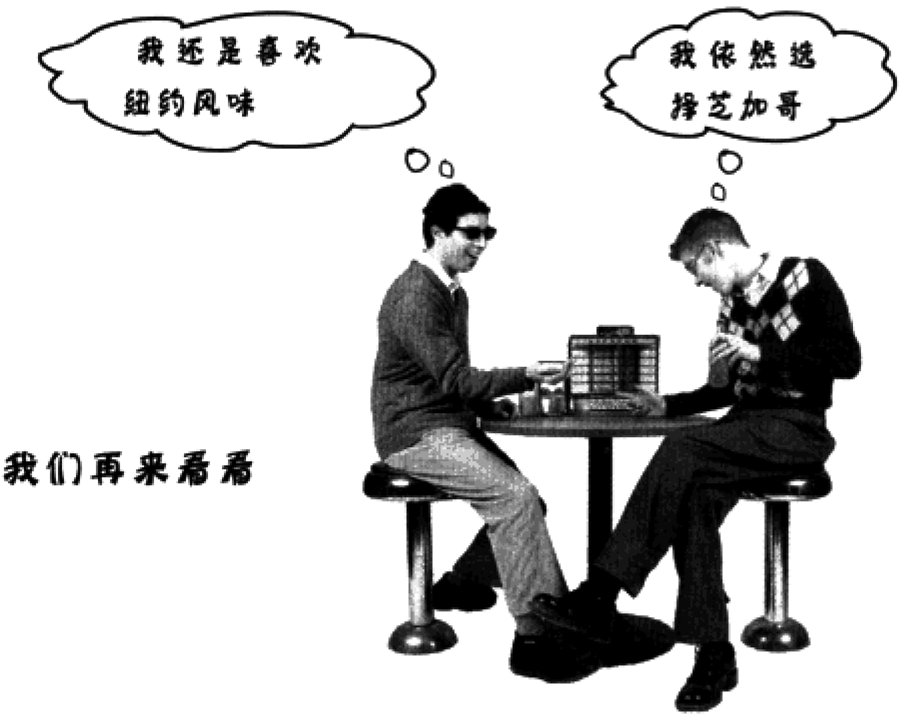
从抽象工厂中派生出一些具体工厂，这些工厂产生相同的产品，但是产品的实现不同。

接着写下我们的代码，然后使用这个工厂来创建产品。通过传入各种不同的工厂，可以制作出各种不同的产品。但是客户代码始终保持不变。

给Ethan和Joel更多的比萨



Ethan和Joel对于对象村的比萨欲罢不能！其实他们不知道，现在所订购的比萨是利用新原料工厂的原料制作出来的。因此当他们订购比萨时……

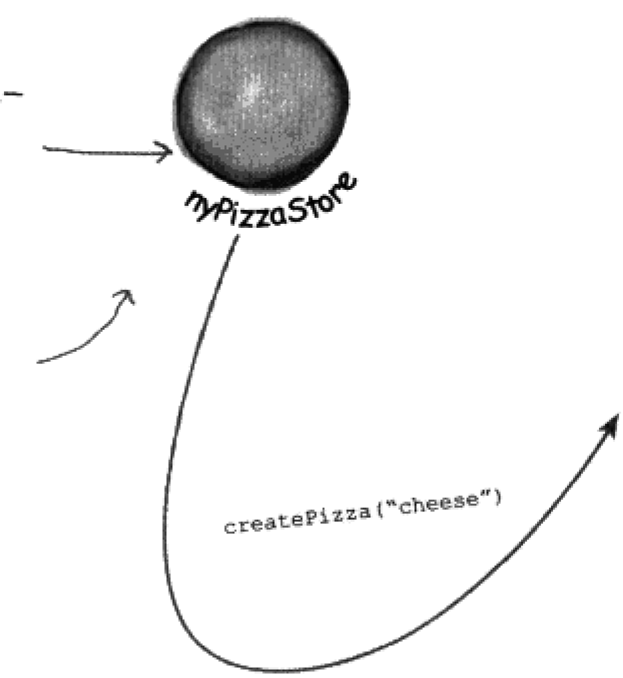


一开始订购的流程依然完全不变，让我们再来看看Ethan的订单：

1 首先我们需要一个纽约比萨店：

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

创建一个NYPizza-Store的实例。



2 现在已经有一个比萨店了，可以接受订单：

```
nyPizzaStore.orderPizza("cheese");
```

调用nyPizzaStore实例的orderPizza()方法。

3 orderPizza()方法首先调用createPizza()方法：

```
Pizza pizza  = createPizza("cheese");
```

接下来，就不一样了，因为我们现在使用了原料工厂

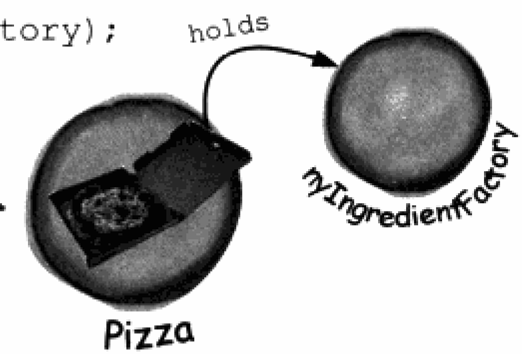
幕后花絮

4 当createPizza()方法被调用时，也就开始涉及原料工厂了：

选择原料工厂，接着在PizzaStore中实例化，然后将它传进每个比萨的构造器中。

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

创建一个比萨的实例，然后将它和纽约原料工厂结合在一起。



5 接下来需要准备比萨。一旦调用了prepare()方法，工厂将被要求准备原料：

prepare()

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

薄饼

大蒜番茄酱料

Reggiano干酪

对Ethan的比萨来说，使用了纽约原料工厂，取得了纽约的原料。

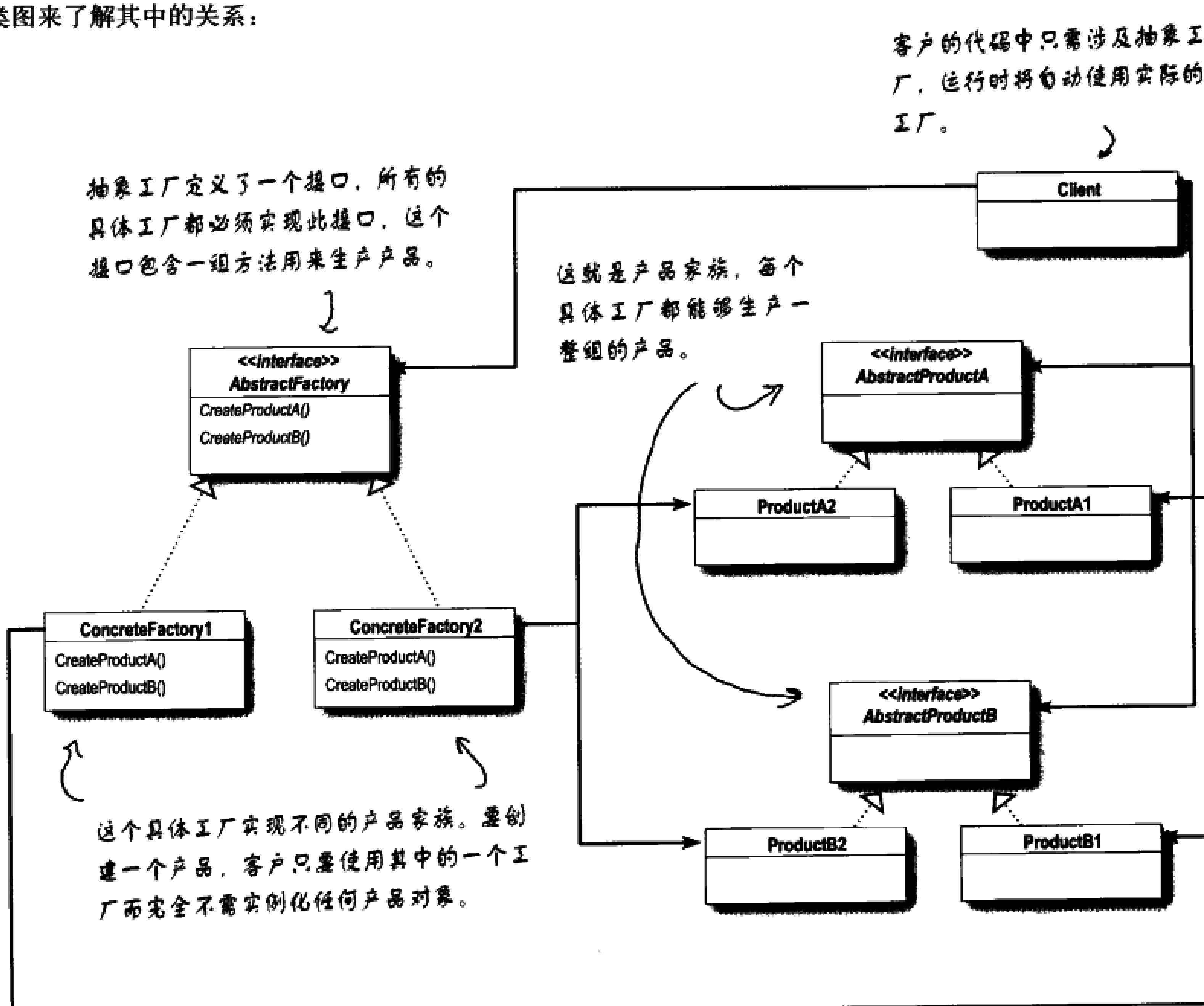
6 最后，我们得到了准备好的比萨，orderPizza()就会接着烘烤、切片、装盒。

定义抽象工厂模式

我们又在模式家族中新增了另一个工厂模式，这个模式可以创建产品的家族。看看这个模式的正式定义：

> **抽象工厂模式**提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。

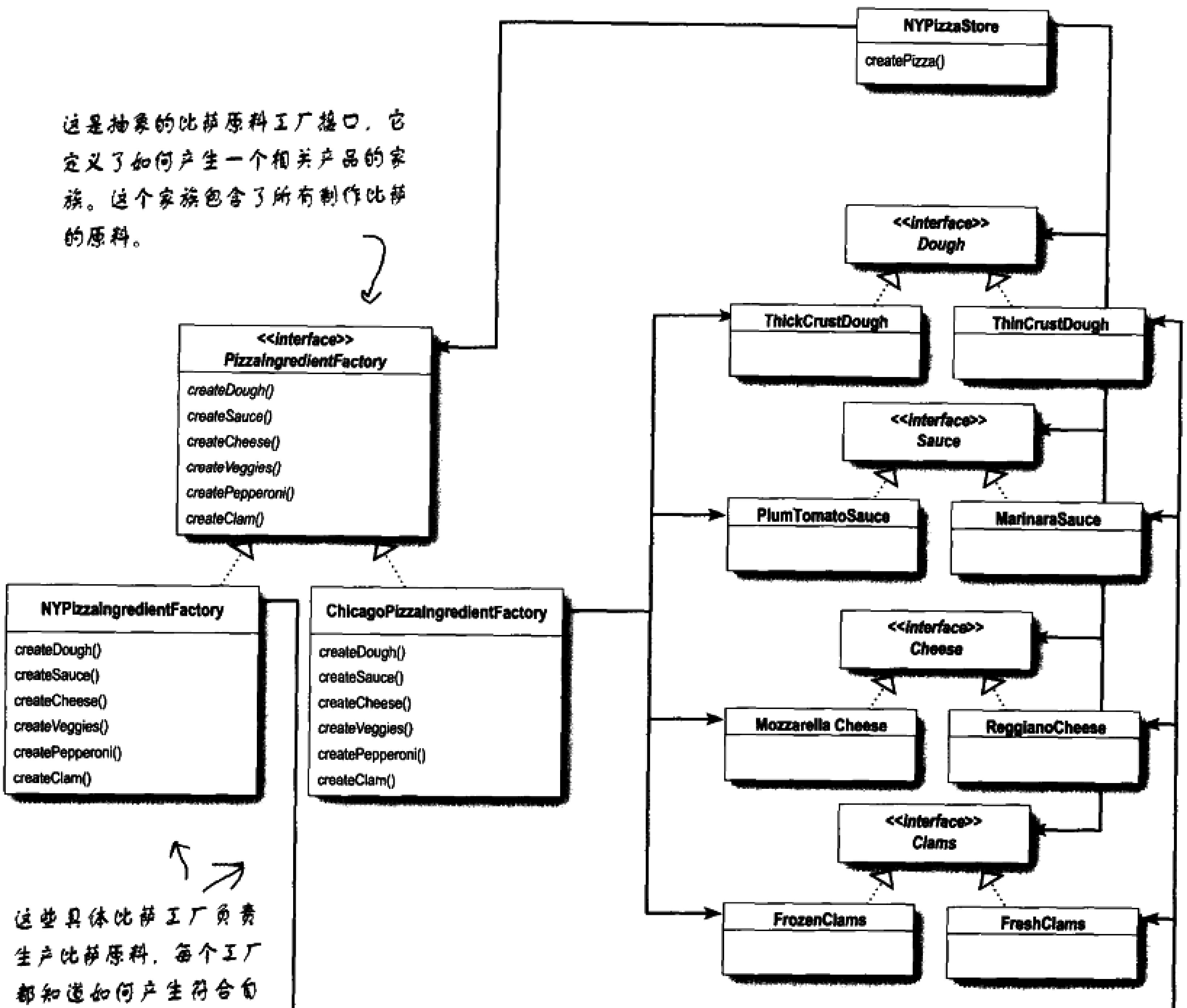
抽象工厂允许客户使用抽象的接口来创建一组相关的产品，而不需要知道（或关心）实际产出的具体产品是什么。这样一来，客户就从具体的产品中被解耦。让我们看看类图来了解其中的关系：



这是一张相当复杂的类图；让我们从PizzaStore的观点来看一看它：

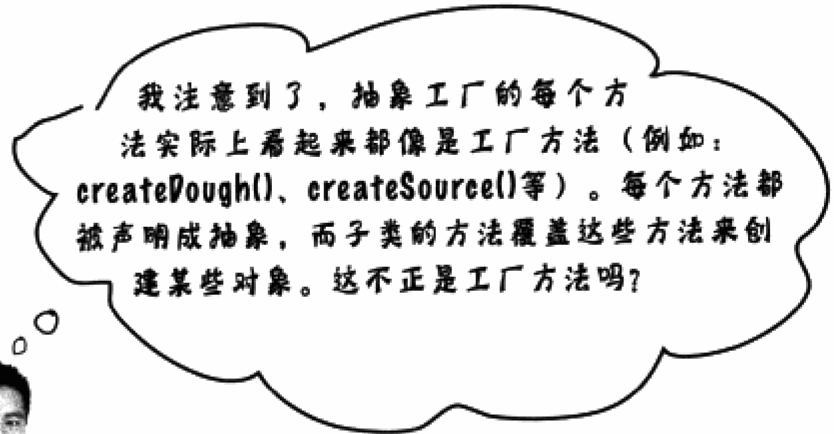
比萨店的两个具体实例（NYPizzaStore、ChicagoPizzaStore）是抽象工厂的客户。

这是抽象的比萨原料工厂接口，它定义了如何产生一个相关产品的家族。这个家族包含了所有制作比萨的原料。



这些具体比萨工厂负责生产比萨原料，每个工厂都知道如何产生符合自己区域的正确对象。

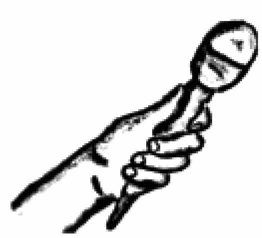
对于这个产品家族，每个工厂都有不同的实现。





工厂方法是不是潜伏在抽象工厂里面？

你的观察力很敏锐！是的，抽象工厂的方法经常以工厂方法的方式实现，这很有道理，对吧？抽象工厂的任务是定义一个负责创建一组产品的接口。这个接口内的每个方法都负责创建一个具体产品，同时我们利用实现抽象工厂的子类来提供这些具体的做法。所以，在抽象工厂中利用工厂方法实现生产方法是相当自然的做法。



模式告白

本周访问：
工厂方法和抽象工厂

HeadFirst：哇！今天很难得，同时请到了两种模式。这可是头一回啊！

工厂方法：呵！我其实不希望人们把我和抽象工厂混为一谈。虽然我们都是工厂模式，但并不表示我们就应该被合在一起访问。

HeadFirst：别生气，我们之所以想要同时采访你们就是为了帮读者搞清楚你们之间谁是谁。你们的确有相似的地方，听说人们常常会把你们搞混了。

抽象工厂：这是真的，有些时候我被错认为是工厂方法。嘿！工厂方法，我知道你也有相同的困扰。我们两个在把应用程序从特定实现中解耦方面真的都很有一套，只是做法不同而已。我能够理解为什么人们总是把我们搞混。

工厂方法：哎呀！这还是让我很不爽。毕竟，我使用的是类而你使用的是对象，根本就不是一回事啊。

HeadFirst：工厂方法，能请你多做一些解释吗？

工厂方法：当然。抽象工厂与我都是负责创建对象，这是我们的工作。但是我用的方法是继承……

抽象工厂：……而我是通过对象的组合。

工厂方法：对！所以这意味着，利用工厂方法创建对象，需要扩展一个类，并覆盖它的工厂方法。

HeadFirst：那这个工厂方法是做什么的呢？

工厂方法：当然是用来创建对象的了。其实整个工厂方法模式，只不过就是通过子类来创建对象。用这种做法，客户只需要知道他们所使用的抽象类型就可以了，而由子类来负责决定具体类型。所以，换句话说，我只负责将客户从具体类型中解耦。

抽象工厂：这一点我也做得到，只是我的做法不同。

HeadFirst：抽象工厂，请继续…… 你刚刚说了一些关于对象组合的事？

抽象工厂：我提供一个用来创建一个产品家族的抽象类型，这个类型的子类定义了产品被产生的方法。要想使用这个工厂，必须先实例化它，然后将它传入一些针对抽象类型所写的代码中。所以，和工厂方法一样，我可以把客户从所使用的实际具体产品中解耦。

HeadFirst：噢！我了解了，所以你的另一个优点是可以把一群相关的产品集合起来。

抽象工厂：对。

HeadFirst：万一需要扩展这组相关产品（比方说新增一个产品），又该怎么办呢？难道这不需要改变接口吗？

抽象工厂：那倒是真的，如果加入新产品就必须改变接口，我知道大家不喜欢这么做……

工厂方法：<窃笑>

抽象工厂：我说，工厂方法，你偷笑什么？

工厂方法：拜托，那可是很严重的！改变接口就意味着必须深入改变每个子类的接口！听起来可是很繁重的工作呀。

抽象工厂：是的，但是我需要一个大的接口，因为我可是被用来创建整个产品家族的。你只不过是创建一个产品，所以你根本不需要一个大的接口，你只需要一个方法就可以了。

HeadFirst：抽象工厂，我听说你经常使用工厂方法来实现你的具体工厂。

抽象工厂：是的，我承认这一点，我的具体工厂经常实现工厂方法来创建他们的产品。不过对我来说，这些具体工厂纯粹只是用来创建产品罢了……

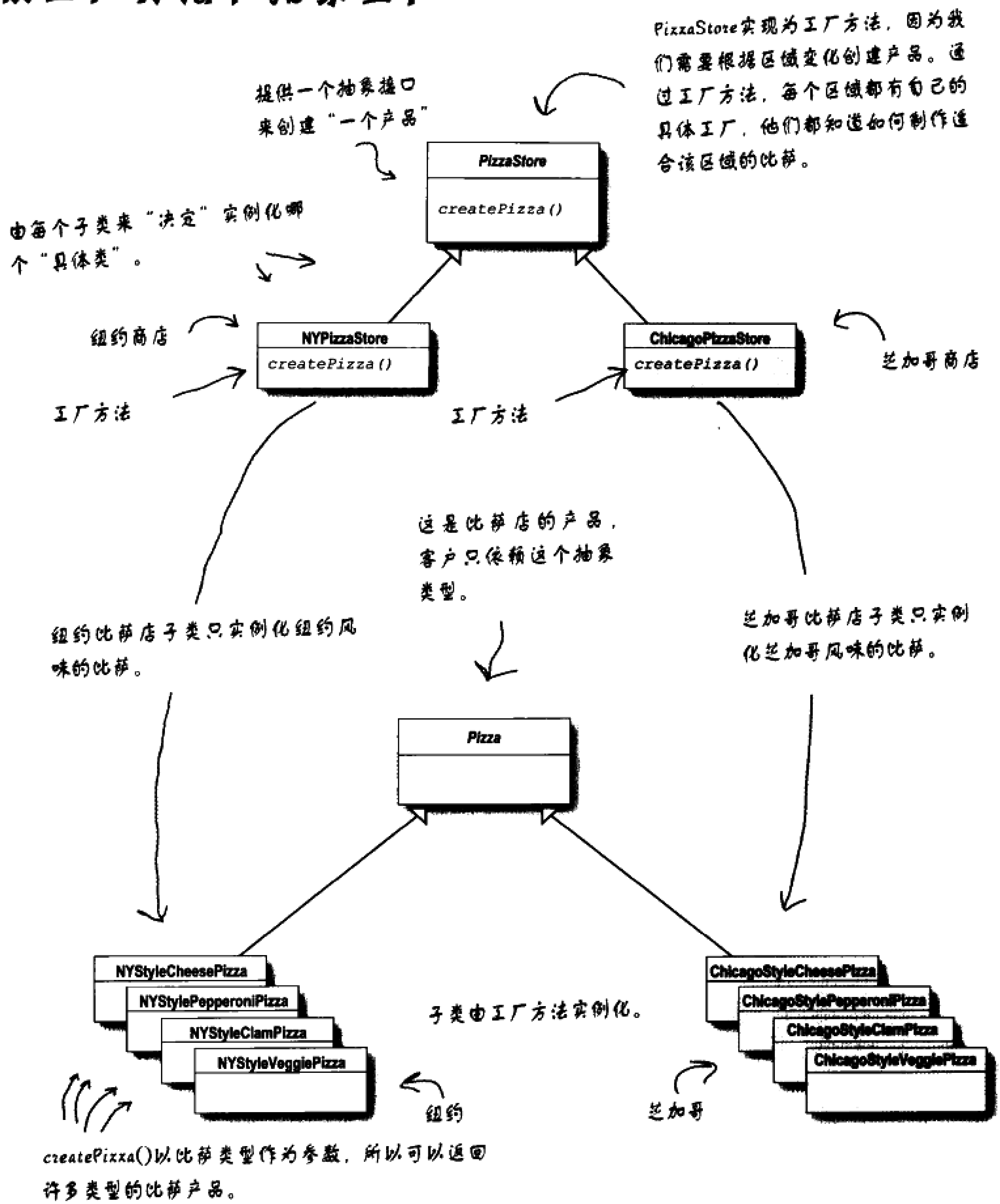
工厂方法：……而对我来说，抽象创建者（creator）中所实现的代码通常会用到子类所创建的具体类型。

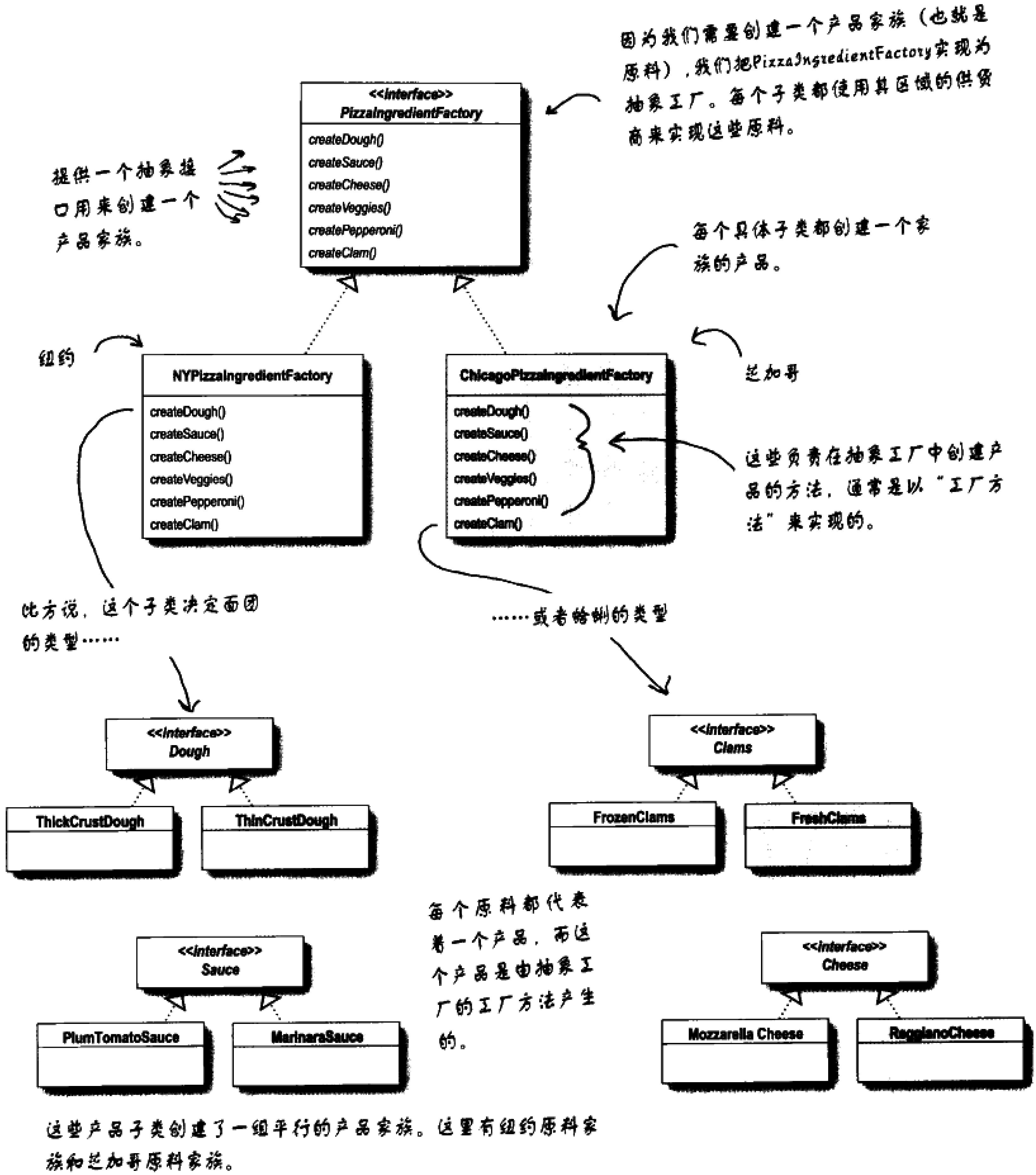
HeadFirst：听起来你们都有自己的一套。我相信人们喜欢有选择的余地，毕竟，工厂这么有用，大家希望在各种不同的情况下都可使用工厂。你们俩都能将对象的创建封装起来，使应用程序解耦，并降低其对特定实现的依赖。真的是很棒。不管是使用工厂方法还是抽象工厂，都可以给人们带来好处。节目结束前，请两位各说几句话吧。

抽象工厂：谢谢。我是抽象工厂，当你需要创建产品家族和想让制造的相关产品集合起来时，你可以使用我。

工厂方法：而我是工厂方法，我可以把你的客户代码从需要实例化的具体类中解耦。或者如果你目前还不知道将来需要实例化哪些具体类时，也可以用我。我的使用方式很简单，只要把我继承成子类，并实现我的工厂方法就可以了。

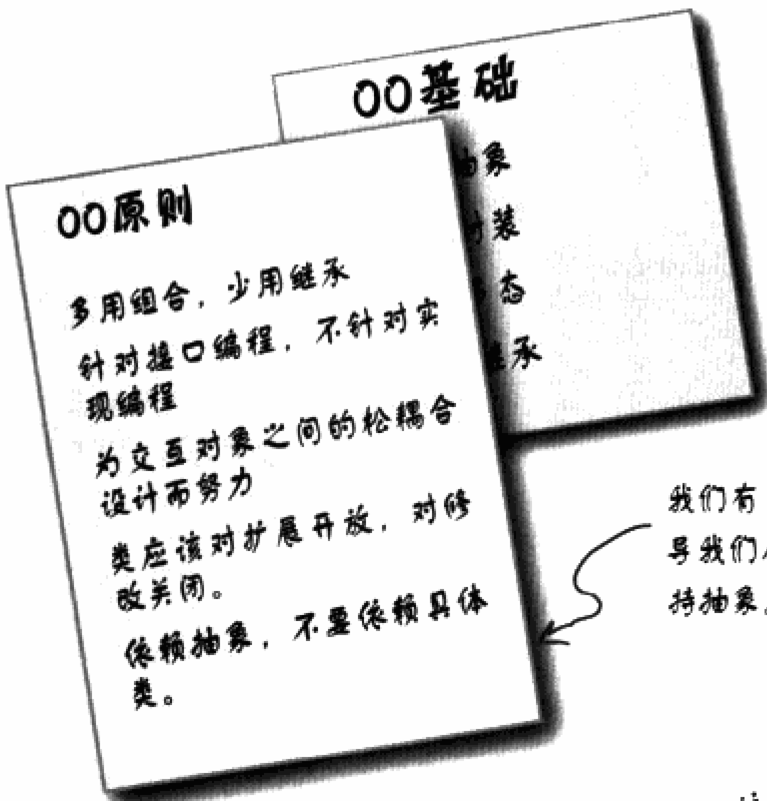
比较工厂方法和抽象工厂



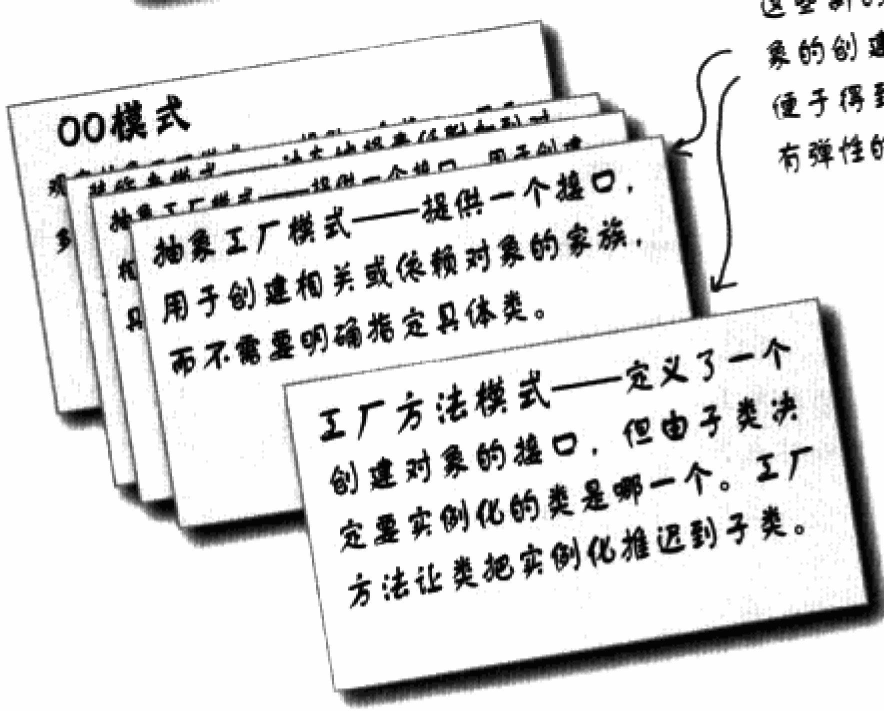

因为我们需要创建一个产品家族（也就是原料），我们把PizzaIngredientFactory实现为抽象工厂。每个子类都使用其区域的供货商来实现这些原料。
<<interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
提供一个抽象接口用来创建一个产品家族。
每个具体子类都创建一个家族的产品。
纽约
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
芝加哥
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
这些负责在抽象工厂中创建产品的方法，通常是以"工厂方法"来实现的。
比方说，这个子类决定面团的类型……
……或者蛤蜊的类型
<<interface>>
Dough
ThickCrustDough
ThinCrustDough
<<interface>>
Clams
FrozenClams
FreshClams
每个原料都代表着一个产品，而这个产品是由抽象工厂的工厂方法产生的。
<<interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
<<interface>>
Cheese
Mozzarella Cheese
ReggianoCheese
这些产品子类创建了一组平行的产品家族。这里有纽约原料家族和芝加哥原料家族。

设计箱内的工具

在本章，我们多加了两个工具到你的工具箱中：工厂方法和抽象工厂。这两种模式都是将对象创建的过程封装起来，以便将代码从具体类解耦。



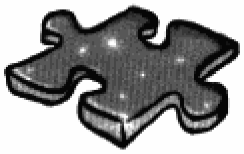
我们有了一个新原则，指导我们尽可能地让事情保持抽象。



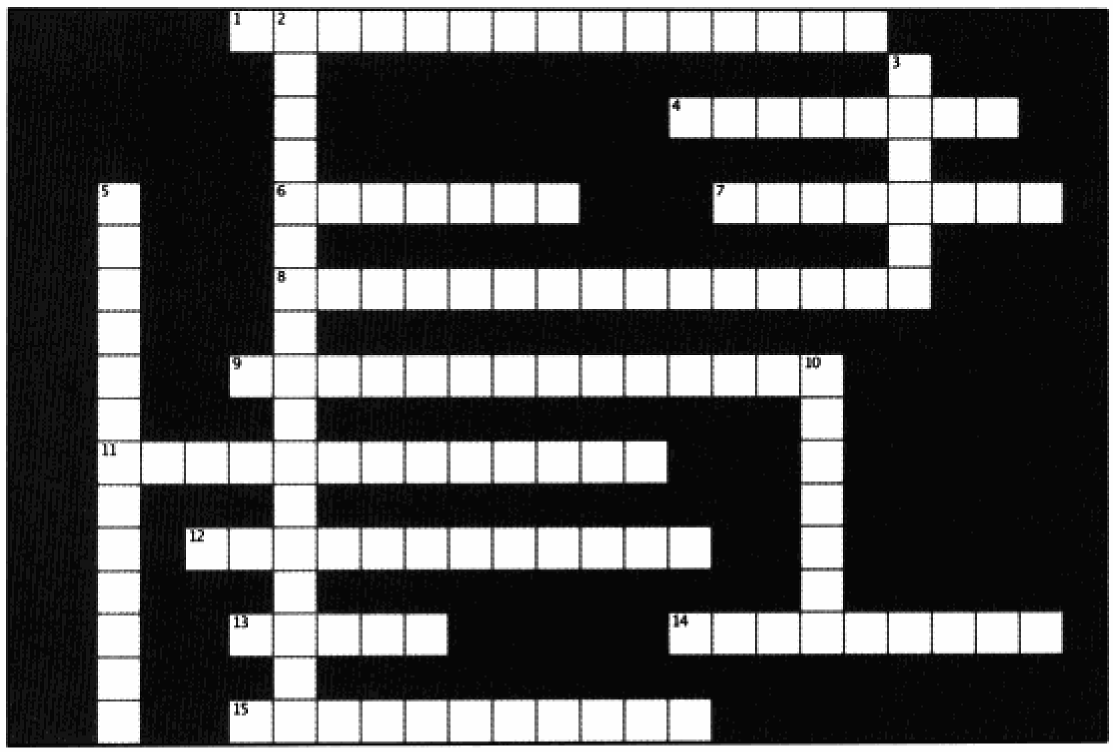
这些新的模式可以将对象的创建封装起来，以便于得到更松耦合、更有弹性的设计。

要点

- 所有的工厂都是用来封装对象的创建
- 简单工厂，虽然不是真正的设计模式，但仍不失为一个简单的方法，可以将客户程序从具体类解耦。
- 工厂方法使用继承：把对象的创建委托给子类，子类实现工厂方法来创建对象。
- 抽象工厂使用对象组合：对象的创建被实现在工厂接口所暴露出来的方法中。
- 所有工厂模式都通过减少应用程序和具体类之间的依赖促进松耦合。
- 工厂方法允许类将实例化延迟到子类进行。
- 抽象工厂创建相关的对象家族，而不需要依赖它们的具体类。
- 依赖倒置原则，指导我们避免依赖具体类型，而要尽量依赖抽象。
- 工厂是很有威力的技巧，帮助我们针对抽象编程，而不要针对具体类编程。



好长的一章呀！让我们边吃比萨边玩拼字游戏，放松片刻吧！答案都是取自本章的英文单词。



横排提示：

1. In Factory Method, each franchise is a ______________
4. In Factory Method, who decides which class to instantiate?
6. Role of PizzaStore in Factory Method Pattern
7. All New York Style Pizzas use this kind of cheese
8. In Abstract Factory, each ingredient factory is a ______________
9. When you use new, you are programming to an ____________
11. createPizza() is a ______________ (two words)
12. Joel likes this kind of pizza
13. In Factory Method, the PizzaStore and the concrete Pizzas all depend on this abstraction
14. When a class instantiates an object from a concrete class, it's ___________ on that object
15. All factory patterns allow us to __________ object creation

竖排提示：

2. We used ______________ in Simple Factory and Abstract Factory and inheritance in Factory Method
3. Abstract Factory creates a ___________ of products
5. Not a REAL factory pattern, but handy nonetheless
10. Ethan likes this kind of pizza



习题解答

Sharpen your pencil

我们已经成功地完成了NYPizzaStore，还剩下实现两个比萨店。就可以开加盟店了。下面是芝加哥和加州的比萨店实现：

这两个比萨店都和纽约店的做法几乎一致……只是创建不同种类的比萨。

```
public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

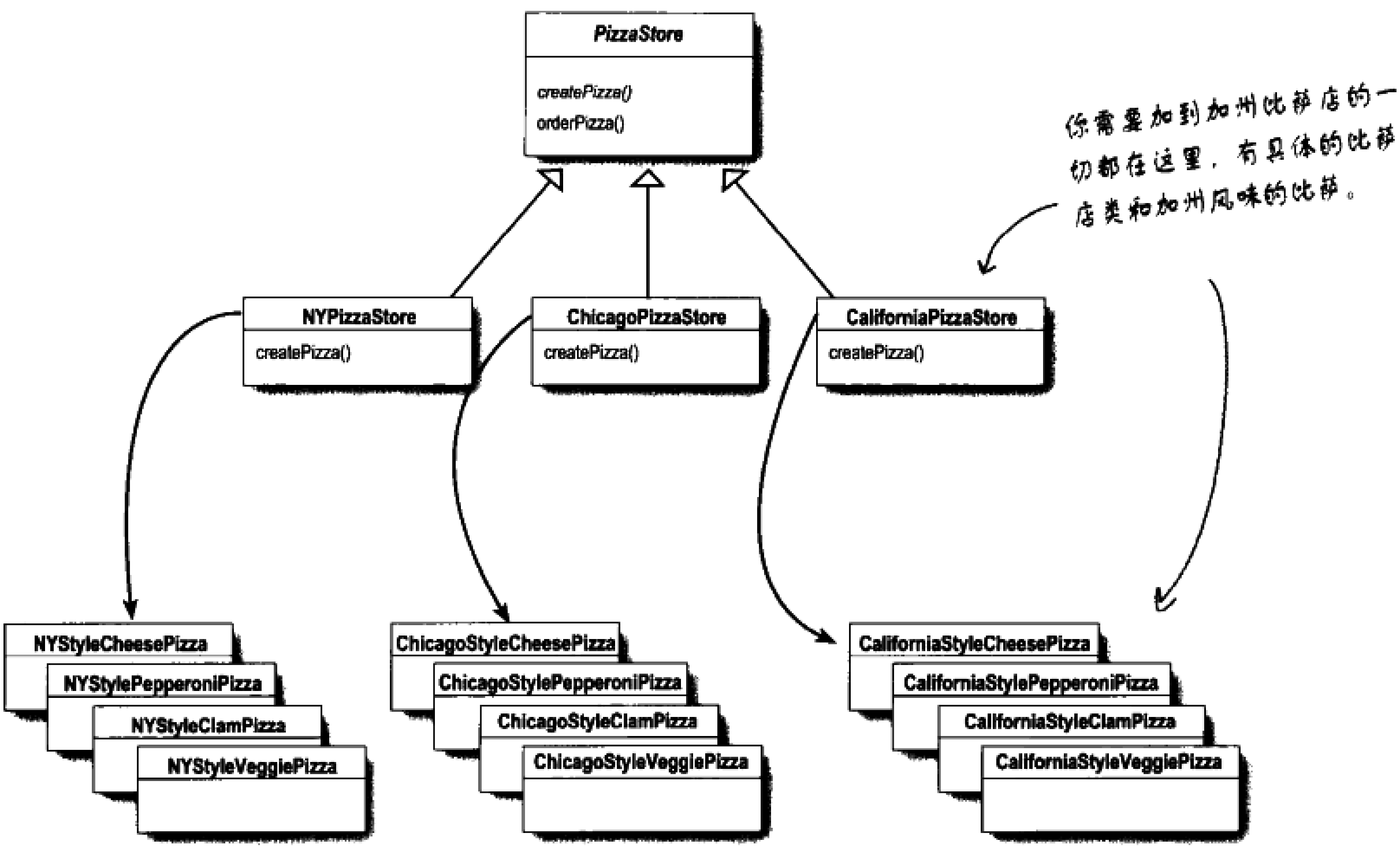
对于芝加哥比萨店，我们只要确认创建芝加哥风味的比萨……

```
public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new CaliforniaStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```

而对于加州比萨店，我们要创建加州风味的比萨。

设计谜题解答

我们需要另一种比萨来符合那些疯狂加州人的需求（当然，这里的疯狂是指好的那一方面）。请绘制出另一组平行的类，把加州区域纳入PizzaStore中。



好了，发挥你的想象力，找出五个“最奇特”的东西加入到比萨中。然后你就可以准备到加州去开比萨店了！

这是我们的建议……

马铃薯泥加烤大蒜

烤肉酱

朝鲜蓟果实

M&M巧克力

花生

一个很依赖的比萨店

Sharpen your pencil

假设你从未听说过OO工厂。下面是一个不使用工厂模式的比萨店版本。数一数，这个类所依赖的具体比萨对象有几种。如果又加了一种加州风味比萨到这个比萨店中，那么届时又会依赖几个对象？

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

处理所有纽约风味比萨

处理所有芝加哥风味比萨

可以把答案写在这里

8 数目

12 加上加州风味后的数目

Sharpen your pencil

写下ChicagoPizzaIngredientFactory的代码。你可以参考下面的类，写出你的实现：

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{
    public Dough createDough() {
        return new ThickCrustDough();
    }

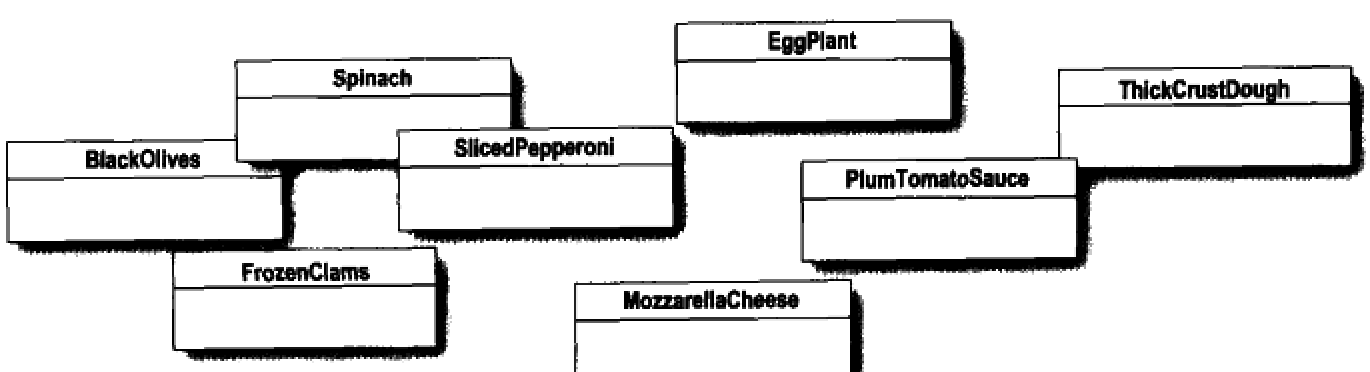
    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

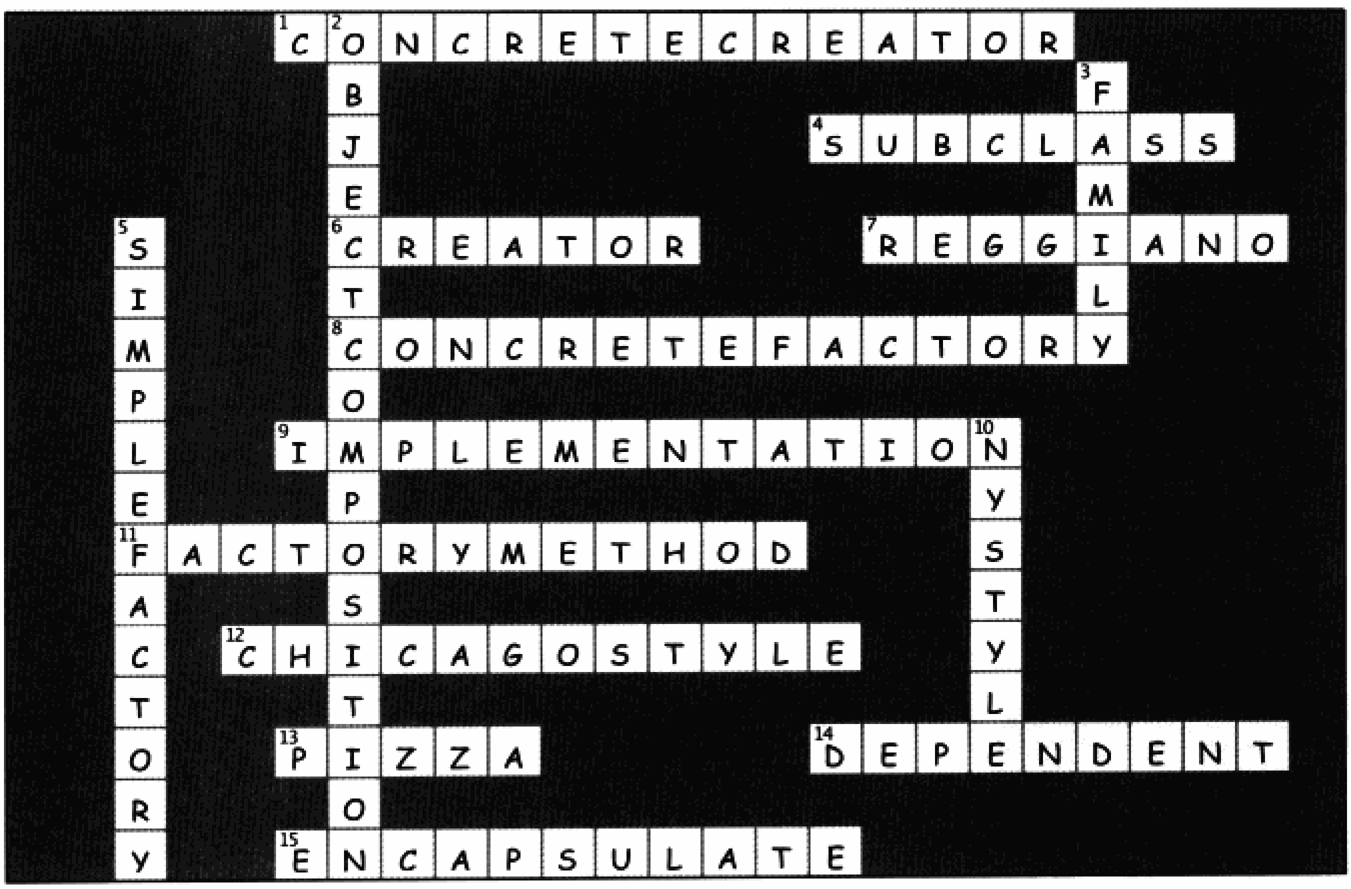
    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                              new Spinach(),
                              new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```



习题解答


1 CONCRETECREATOR
2 OBJECTCOMPOSITION
3 FAMILY
4 SUBCLASS
5 SIMPLEFACTORY
6 CREATOR
7 REGGIANO
8 CONCRETEFACTORY
9 IMPLEMENTATION
10 NYSTYL
11 FACTORYMETHOD
12 CHICAGOSTYLE
13 PIZZA
14 DEPENDENT
15 ENCAPSULATE

5 单件模式

独一无二的对象



下一站是单件模式（Singleton Pattern）：用来创建独一无二的,只能有一个实例的对象的入场券。告诉你一个好消息，单件模式的类图可以说是所有模式的类图中最简单的，事实上，它的类图上只有一个类！但是，可不要兴奋过头，尽管从类设计的视角来说它很简单，但是实现上还是会遇到相当多的波折。所以，系好安全带，出发了！





开发人员：这有什么用处？

大师：有一些对象其实我们只需要一个，比方说：线程池（threadpool）、缓存（cache）、对话框、处理偏好设置和注册表（registry）的对象、日志对象，充当打印机、显卡等设备的驱动程序的对象。事实上，这类对象只能有一个实例，如果制造出多个实例，就会导致许多问题产生，例如：程序的行为异常、资源使用过量，或者是不一致的结果。

开发人员：好吧！或许的确有一些类应该只存在一个实例，但这需要花整个章节的篇幅来说明吗？难道不能靠程序员之间的约定或是利用全局变量做到？你知道的，利用Java的静态变量就可以做到。

大师：许多时候，的确通过程序员之间的约定就可以办到。但如果有更好的做法，大家应该都乐意接受。别忘了，就跟其他的模式一样，单件模式是经得起时间考验的方法，可以确保只有一个实例会被创建。单件模式也给了我们一个全局的访问点，和全局变量一样方便，又没有全局变量的缺点。

开发人员：什么缺点？

大师：举例来说：如果将对象赋值给一个全局变量，那么你必须在程序一开始就创建好对象★，对吧？万一这个对象非常耗费资源，而程序在这次的执行过程中又一直没用到它，不就形成浪费了吗？稍后你会看到，利用单件模式，我们可以在需要时才创建对象。

开发人员：我还是觉得这没什么困难的。

大师：利用静态类变量、静态方法和适当的访问修饰符（access modifier），你的确可以做到这一点。但是，不管使用哪一种方法，能够了解单件的运作方式仍然是很有趣的事。单件模式听起来简单，要做得对可不简单。不信问问你自己：要如何保证一个对象只能被实例化一次？答案可不是三言两语就说得完的，是不是？

★这其实和实现有关。有些JVM的实现是：在用到的时候才创建对象。

小小单件

苏格拉底式的诱导问答

如何创建一个对象？	`new MyObject();`
万一另一个对象想创建MyObject会怎样？可以再次new MyObject吗？	是的，当然可以。
所以，一旦有一个类，我们是否都能多次地实例化它？	如果是公开的类，就可以。
如果不是的话，会怎样？	如果不是公开类，只有同一个包内的类可以实例化它，但是仍可以实例化它多次。
嗯！有意思！你知道可以这么做吗？	我没想过。但是，这是合法的定义，有一定的道理。

```
public MyClass {

   private MyClass() {}

}
```

怎么说呢？	我认为含有私有的构造器的类不能被实例化。
有可以使用私有的构造器的对象吗？	嗯，我想MyClass内的代码是唯一能调用此构造器的代码。但是这又不太合乎常理。

为什么？

因为必须有MyClass类的实例才能调用MyClass构造器，但是因为没有其他类能够实例化MyClass，所以我们得不到这样的实例。这是“鸡生蛋，蛋生鸡”的问题。我可以在MyClass类型的对象上使用MyClass构造器，但是在这之前，必须有一个MyClass实例。在产生MyCalss实例之前，又必须在MyClass实例内才能调用私有的构造器……

嘿！我有个想法。
你认为这样如何？

```
public MyClass {

   public static MyClass getInstance() {
   }
}
```

MyClass有一个静态方法。我们可以这样调用这个方法：

```
MyClass.getInstance();
```

为何调用的时候用MyClass的类名，而不是用对象名？

因为getInstance()是一个静态方法，换句话说，是一个“类”方法。引用一个静态方法，你需要使用类名。

有意思。假如把这些合在一起“是否”就可以初始化一个MyClass？

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

当然可以。

好了，你能想出第二种实例化对象的方式吗？

```
MyClass.getInstance();
```

你能够完成代码使MyClass只有一个实例被产生吗？

嗯，大概可以吧……

（下一页有这个代码。）

剖析经典的单件模式实现

把MyClass改名为Singleton。

利用一个静态变量来记录Singleton类的唯一实例。

```
public class Singleton {
        private  static  Singleton
uniqueInstance;
    // 这里是其他的有用实例化变量

    private Singleton() {}

        public  static  Singleton
getInstance() {
        if (uniqueInstance == null)
{
              uniqueInstance  =  new
Singleton();
        }
        return uniqueInstance;
    }

    // 这里是其他的有用方法
}
```

把构造器声明为私有的，只有自Singleton类内才可以调用构造器。

用getInstance()方法实例化对象，并返回这个实例。

当然，Singleton是一个正常的类，具有一些其他用途的实例变量和方法。

注意！

如果你只是很快地翻到这一页，不要盲目地键入代码。在本章后面的部分中，你会看到这个版本有一些问题。

再靠近一点

uniqueInstance拥有“一个”实例，别忘了，它是个静态变量。

如果uniqueInstance是空的，表示还没有创建实例……

……而如果它不存在，我们就利用私有的构造器产生一个Singleton实例并把它赋值到uniqueInstance静态变量中。请注意，如果我们不需要这个实例，它就永远不会产生。这就是“延迟实例化”（lazy instantiaze）。

```
if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;
```

如果uniqueInstance不是null，就表示之前已经创建过对象。我们就直接跳到return语句。

当执行到这个return，就表示我们已经有了实例，并将uniqueInstance当返回值。



模式告白

本周访问：
单件的告白

HeadFirst：今天我们很高兴专访单件对象。一开始，不妨先介绍一下你自己。

单件：关于我，我只能说我很独特，我是独一无二的。

HeadFirst：独一无二？

单件：是的，独一无二。我是利用单件模式构造出来的，这个模式让我在任何时刻都只有一个对象。

HeadFirst：这样不会有点浪费吗？毕竟有人花了这么多时间写了类的代码，而这个类竟然只产生一个对象。

单件：不，一点儿也不浪费！“一个”的威力很强大呢！比方说，如果有一个注册表设置（registry setting）的对象，你不希望这样的对象有多个拷贝吧？那会把设置搞得一团乱。利用像我这样的单件对象，你可以确保程序中使用的全局资源只有一份。

HeadFirst：请继续……

单件：嗯！我擅长许多事。有时候独身是有些好处的。我常常被用来管理共享的资源，例如数据库连接或者线程池。

HeadFirst：但我还是觉得，一个人好像有一点孤单。

单件：因为只有我一个人，所以通常很忙，但还是希望更多开发人员能认识我。许多开发人员因为产生了太多同一类的对象而使他们的代码出现了bug，但他们却浑然不觉。

HeadFirst：那么，请允许我这么问，你怎么能确定只有一个你？说不定别人也会利用new产生多个你呢。

单件：不可能，我是独一无二的。

HeadFirst：该不会要每个开发人员都发毒誓绝对不会实例化多个你吧？

单件：当然不是，真相是…… 唉呀！这牵扯到个人隐私…… 其实…… 我没有公开的构造器。

HeadFirst：没有公开的构造器！！噢！抱歉！我太激动了。没有公开的构造器？

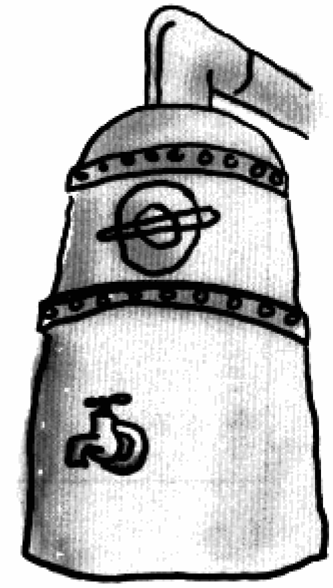
单件：是的，我的构造器是声明为私有的。

HeadFirst：这怎么行得通？你“究竟”是怎样被实例化的？

单件：外人为了要取得我的实例，他们必须“请求”得到一个实例，而不是自行实例化得到一个实例。我的类有一个静态方法，叫做getInstance()。调用这个方法，我就立刻现身，随时可以工作。事实上，我可能是在这次调用的时候被创建出来的，也可能是以前早就被创建出来了。

HeadFirst：单件先生，你的内在比外表更加深奥。谢谢你如此坦白，希望能很快再与你见面。

巧克力工厂



大家都知道，现代化的巧克力工厂具备计算机控制的巧克力锅炉。锅炉做的事，就是把巧克力和牛奶融在一起，然后送到下一个阶段，以制造成巧克力棒。

这里有一个Choc-O-Holic公司的工业强度巧克力锅炉控制器。看看它的代码，你会发现代码写得相当小心，他们在努力防止不好的事情发生。例如：排出500加仑的未煮沸的混合物，或者锅炉已经满了还继续放原料，或者锅炉内还没放原料就开始空烧。

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }
```

代码开始时，锅炉是空的。

```
    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // 在锅炉内填满巧克力和牛奶的混合物
        }
    }
```

在锅炉内填入原料时，锅炉必须是空的。一旦填入原料，就把empty和boiled标志设置好。

```
    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // 排出煮沸的巧克力和牛奶
            empty = true;
        }
    }
```

锅炉排出时，必须是满的（不可以是空的）而且是煮过的。排出完毕后，把empty标志设回true。

```
    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // 将炉内物煮沸
            boiled = true;
        }
    }
```

煮混合物时，锅炉必须是满的，并且是没有煮过的。一旦煮沸后，就把boiled标志设为true。

```
    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

BRAIN POWER

Choc-O-Holic公司在有意识地防止不好的事情发生，你不这么认为吗？你可能会担心，如果同时存在两个ChocolateBoiler（巧克力锅炉）实例，可能将发生很糟糕的事情。

万一同时有多于一个的ChocolateBoiler（巧克力锅炉）实例存在，可能发生哪些很糟糕的事呢？

Sharpen your pencil

请帮Choc-O-Holic改进ChocolateBoiler类，把这个类设计成单件。

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;



           ChocolateBoiler() {
        empty = true;
        boiled = false;
    }




    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // 在锅炉内填充巧克力和牛奶的混合物
        }
    }
    // 其他的部分省略不列出来
}
```

定义单件模式

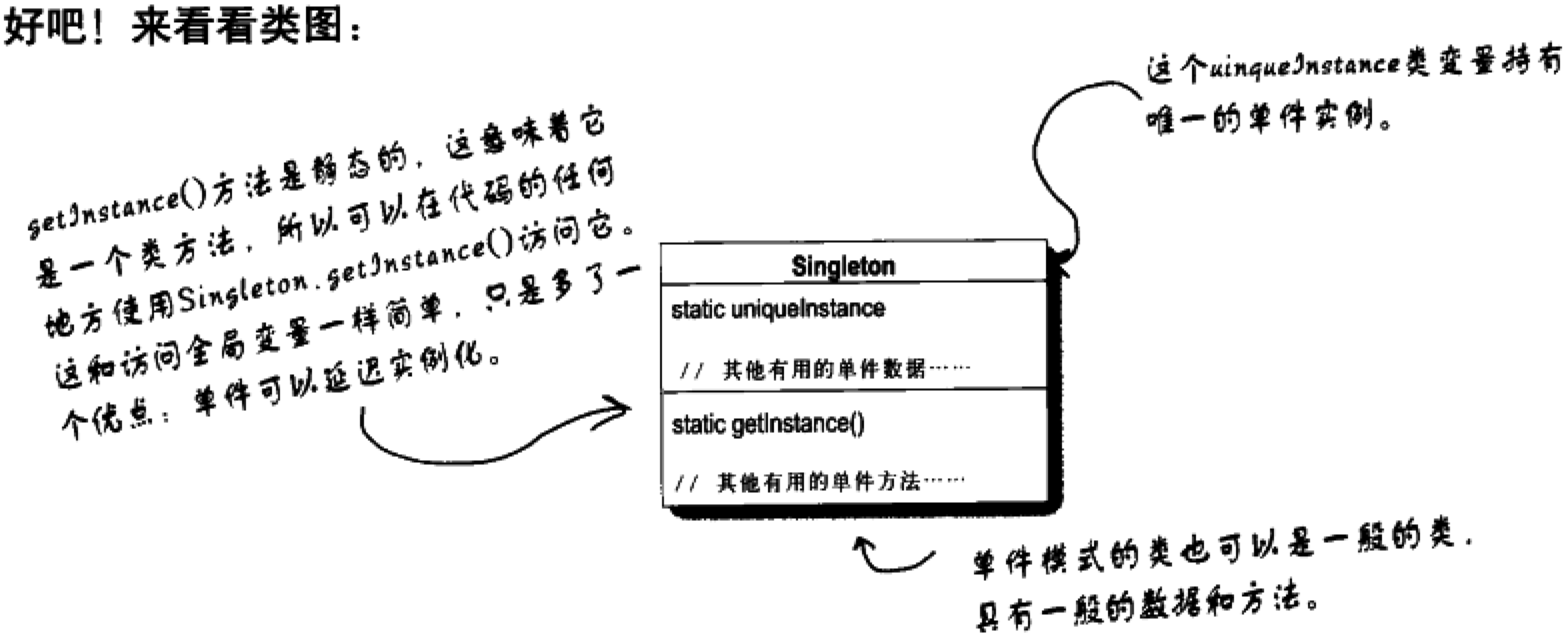
现在你脑海中已经有了单件的经典实现，该是坐下来享受一条巧克力棒，并细细品味单件模式的时候了。
先看看单件模式的简要定义：

> **单件模式**确保一个类只有一个实例，并提供一个全局访问点。

这定义一点儿都不让人吃惊，但是让我们更深入一点儿：

- 到底怎么回事？我们正在把某个类设计成自己管理的一个单独实例，同时也避免其他类再自行产生实例。要想取得单件实例，通过单件类是唯一的途径。
- 我们也提供对这个实例的全局访问点：当你需要实例时，向类查询，它会返回单个实例。前面的例子利用延迟实例化的方式创建单件，这种做法对资源敏感的对象特别重要。

好吧！来看看类图：



Hershey ~~Houston~~，我们遇到麻烦了……

看起来巧克力锅炉要让我们失望了，尽管我们利用经典的单件来改进代码，但是ChocolateBoiler的fill()方法竟然允许在加热的过程中继续加入原料。这可是会溢出五百加仑的原料（牛奶和巧克力）呀！怎么会这样！？





多加线程，就会造成这样吗？不是只要为ChocolateBolier的单件设置好uniqueInstance变量，所有的getInstance()调用都会取得相同的实例吗？对不对？

化身为JVM



这里有两个线程都要执行这段代码。你的工作是扮演JVM角色并判断出两个线程是否可能抓住不同的锅炉对象而扰乱这段代码。提示：你只需要检查getInstance()方法内的操作次序和uniqueInstance的值，看它们是否互相重叠。

用代码帖来帮你研究这段代码为什么可能产生两个锅炉对象。

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

```
public static ChocolateBoiler
      getInstance() {
```

```
if (uniqueInstance == null) {
```

```
uniqueInstance =
    new ChocolateBoiler();
```

```
}
```

```
return uniqueInstance;
```

```
}
```

记得在翻页前，先看看188页的答案。

线程一	线程二	uniqueInstance的值

处理多线程

只要把getInstance()变成同步（synchronized）方法，多线程灾难几乎就可以轻易地解决了：

```
public class Singleton {
    private static Singleton uniqueInstance;

    // 其他有用的实例化的变量
    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // 其他有用的方法
}
```

通过增加synchronized关键字到getInstance()方法中，我们迫使每个线程在进入这个方法之前，要先等候别的线程离开该方法。也就是说，不会有两个线程可以同时进入这个方法。



说得很对，的确是有一点不好。而比你所想象的还要严重一些的是：只有第一次执行此方法时，才真正需要同步。换句话说，一旦设置好uniqueInstance变量，就不再需要同步这个方法了。之后每次调用这个方法，同步都是一种累赘。

能够改善多线程吗？

为了要符合大多数Java应用程序，很明显地，我们需要确保单件模式能在多线程的状况下正常工作。但是似乎同步getInstance()的做法将拖垮性能，该怎么办呢？

可以有一些选择……

1．如果getInstance()的性能对应用程序不是很关键，就什么都别做

没错，如果你的应用程序可以接受getInstance()造成的额外负担，就忘了这件事吧。同步getInstance()的方法既简单又有效。但是你必须知道，同步一个方法可能造成程序执行效率下降100倍。因此，如果将getInstance()的程序使用在频繁运行的地方，你可能就得重新考虑了。

2．使用"急切"创建实例，而不用延迟实例化的做法

如果应用程序总是创建并使用单件实例，或者在创建和运行时方面的负担不太繁重，你可能想要急切（eagerly）创建此单件，如下所示：

在静态初始化器（static initializer）中创建单件。这段代码保证了线程安全（thread safe）。

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

已经有实例了，直接使用它。

利用这个做法，我们依赖JVM在加载这个类时马上创建此唯一的单件实例。JVM保证在任何线程访问uniqueInstance静态变量之前，一定先创建此实例。

3.用“双重检查加锁”，在getInstance()中减少使用同步

利用双重检查加锁（double-checked locking），首先检查是否实例已经创建了，如果尚未创建，“才”进行同步。这样一来，只有第一次会同步，这正是我们想要的。

来看看代码：

```
public class Singleton {
    private volatile* static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

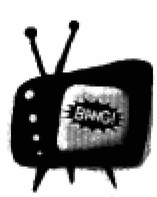
检查实例，如果不存在，就进入同步区块。

注意，只有第一次才彻底执行这里的代码。

进入区块后，再检查一次。如果仍是null，才创建实例。

★ volatile关键词确保：当uniqueInstance变量被初始化成Singleton实例时，多个线程正确地处理uniqueInstance变量。

如果性能是你关心的重点，那么这个做法可以帮你大大地减少getInstance()的时间耗费。



注意！

双重检查加锁不适用于1.4及更早版本的Java！

很不幸地，在1.4及更早版本的Java中，许多JVM对于volatile关键字的实现会导致双重检查加锁的失效。如果你不能使用Java 5，而必须使用旧版的Java，就请不要利用此技巧实现单件模式。

再度回到巧克力工厂……



在研究如何摆脱多线程的梦魇同时，巧克力锅炉也被清理干净可以再度开工了。首先，得处理多线程的问题。我们有一些选择方案，每个方案都有优缺点，到底该采用哪一个？



描述每一种方案对于修改巧克力锅炉代码所遇到的问题的适用性。

同步getInstance()方法：

____

____

急切实例化

____

____

双重检查加锁

____

____

恭喜！

此刻，巧克力工厂的问题已经解决了，而且Choc-O-Holic很高兴在锅炉的代码中能够采用这些专业知识。不管你使用哪一种多线程解决方案，锅炉都能顺畅工作，不会有闪失。恭喜你，不但避免了500磅热巧克力的危机，也认清了单件所带来的所有潜在问题。

there are no Dumb Questions

问： 单件模式只有一个类，应该是很简单的模式，但是问题似乎不少。

答： 唉呀！我们只是提前警告，读者不要因为这点儿问题而泄气。固然正确地实现单件模式需要一点技巧，但是在阅读完本章之后，你已经具备了用正确的方法实现单件模式的能力。当你需要控制实例个数时，还是应当使用单件模式。

问： 难道我不能创建一个类，把所有的方法和变量都定义为静态的，把类直接当做一个单件？

答： 如果你的类自给自足，而且不依赖于复杂的初始化，那么你可以这么做。但是，因为静态初始化的控制权是在Java手上，这么做有可能导致混乱，特别是当有许多类牵涉其中的时候。这么做常常会造成一些微妙的、不容易发现的和初始化的次序有关的bug。除非你有绝对的必要使用类的单件，否则还是建议使用对象的单件，比较保险。

问： 那么类加载器（class loader）呢？听说两个类加载器可能有机会各自创建自己的单件实例。

答： 是的。每个类加载器都定义了一个命名空间，如果有两个以上的类加载器，不同的类加载器可能会加载同一个类，从整个程序来看，同一个类会被加载多次。如果这样的事情发生在单件上，就会产生多个单件并存的怪异现象。所以，如果你的程序有多个类加载器又同时使用了单件模式，请小心。有一个解决办法：自行指定类加载器，并指定同一个类加载器。

放轻松

谣传垃圾收集器会吃掉单件，这过分夸大了！

在Java 1.2之前，垃圾收集器有个bug，会造成当单件在没有全局的引用时被当作垃圾清除。也就是说，如果一个单件只有本单件类引用它本身，那么该单件就会被当做垃圾清除。这造成让人困惑的bug：因为在单件被清除之后，下次调用getInstance()会产生一个“全新的”单件。对很多程序来说，这会造成让人困惑的行为，因为对象的实例变量值都不见了，一切回到最原始的设置（例如：网络连接被重新设置）。

Java 1.2以后，这个bug已经被修正了，也不再需要一个全局引用来保护单件。如果出于某些原因你还在用旧版的Java，要特别注意这个问题。如果你使用1.2以后的Java，就可以高枕无忧了。

问： **我所受到的教育一直是：类应该做一件事，而且只做一件事。类如果能做两件事，就会被认为是不好的OO设计。单件有没有违反这样的观念呢？**

答： 你说的是"一个类，一个责任"原则。没错，你是对的。单件类不只负责管理自己的实例（并提供全局访问），还在应用程序中担当角色，所以也可以被视为是两个责任。尽管如此，由类管理自己的实例的做法并不少见。这可以让整体设计更简单。更何况，许多开发人员都已经熟悉了单件模式的这种做法。

问： **我想把单件类当成超类，设计出子类，但是我遇到了问题：究竟可以不可以继承单件类？**

答： 继承单件类会遇到的一个问题，就是构造器是私有的。你不能用私有构造器来扩展类。所以你必须把单件的构造器改成公开的或受保护的。但是这么一来就不算是"真正的"单件了，因为别的类也可以实例化它。

如果你果真把构造器的访问权限改了，还有另一个问题会出现。单件的实现是利用静态变量，直接继承会导致所有的派生类共享同一个实例变量，这可能不是你想要的。所以，想要让子类能工作顺利，基类必须实现注册表（Registry）功能。

在这么做之前，你得想想，继承单件能带来什么好处。就和大多数的模式一样，单件不一定适合设计进入一个库中。而且，任何现有的类，都可以轻易地加上一些代码支持单件模式。最后，如果你的应用程序大量地使用了单件模式，那么你可能需要再好好地检查你的设计。因为通常适合使用单件模式的机会不多。

问： **我还是不了解为何全局变量比单件模式差。**

答： 在Java中，全局变量基本上就是对对象的静态引用。在这样的情况下使用全局变量会有一些缺点，我们已经提到了其中的一个：急切实例化VS.延迟实例化。但是我们要记住这个模式的目的：确保类只有一个实例并提供全局访问。全局变量可以提供全局访问，但是不能确保只有一个实例。全局变量也会变相鼓励开发人员，用许多全局变量指向许多小对象来造成命名空间（namespace）的污染。单件不鼓励这样的现象，但单件仍然可能被滥用。



设计箱内的工具

你又加了一个新的模式到工具箱里。单件提供另一种创建对象的方法，创建独一无二的对象。

OO基础

抽象
封装
多态
继承

OO原则

封装变化
多用组合，少用继承
针对接口编程，不针对实现编程
为交互对象之间的松耦合设计而努力
类应该对扩展开放，对修改关闭。
依赖抽象，不要依赖具体类。

OO模式

单件模式——确保一个类只有一个实例，并提供全局访问点。

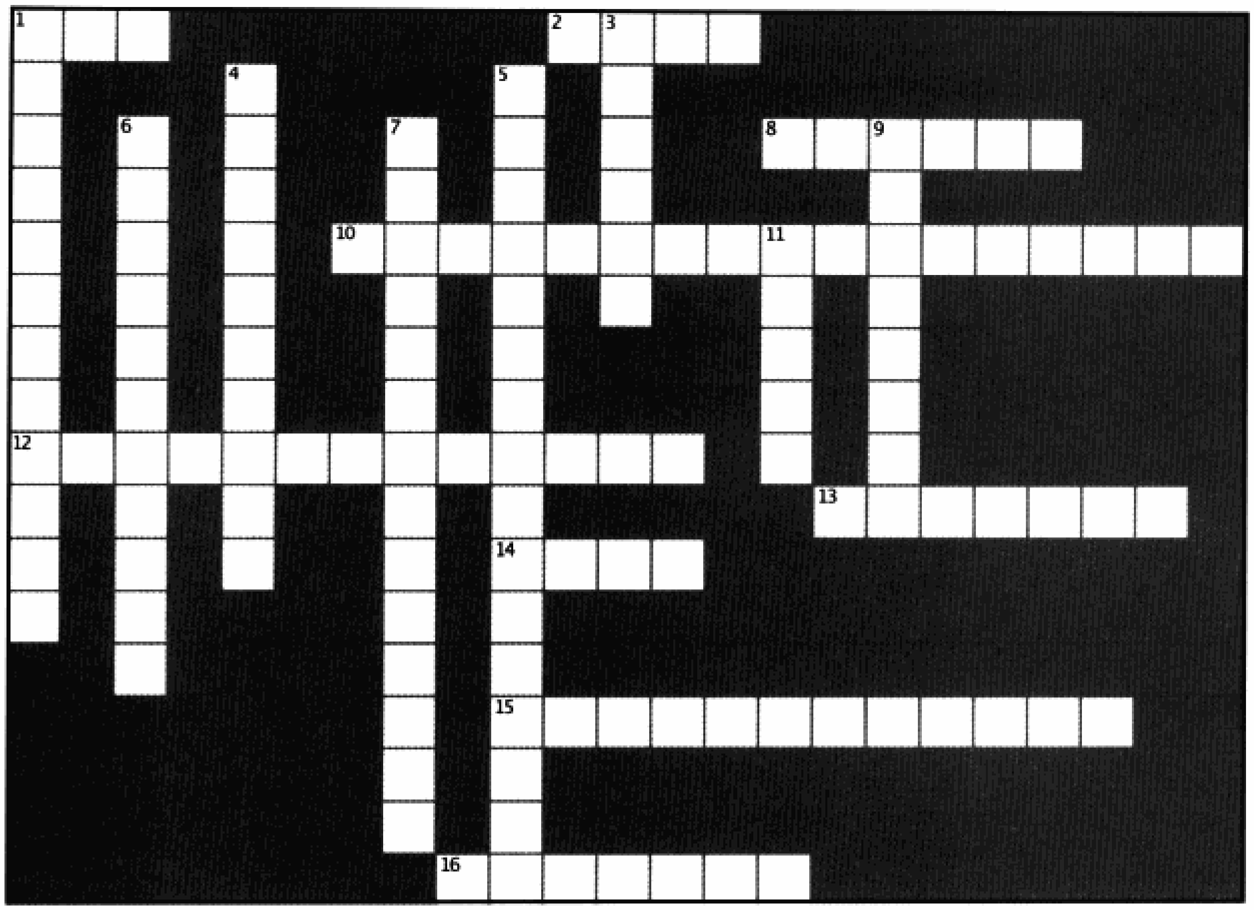
当你需要确保程序中的某个类只有一个实例时，就采用单件模式吧！

正如你所看到的，尽管看起来很简单，但单件实现中涉及到了很多细节。读完本章后，你就可以使用单件了。

要点

- 单件模式确保程序中一个类最多只有一个实例。
- 单件模式也提供访问这个实例的全局点。
- 在Java中实现单件模式需要私有的构造器、一个静态方法和一个静态变量。
- 确定在性能和资源上的限制，然后小心地选择适当的方案来实现单件，以解决多线 程的问题（我们必须认定所有的程序都是多线程的）。
- 如果不是采用第五版的Java 2，双重检查加锁实现会失效。
- 小心，你如果使用多个类加载器，可能导致单件失效而产生多个实例。
- 如果使用JVM 1.2或之前的版本，你必须建立单件注册表，以免垃圾收集器将单件回收。

坐下来，打开因为解决多线程问题而获赠的巧克力，花一点儿时间解决这个填字游戏，所有的答案都是来自本章的英文词汇。



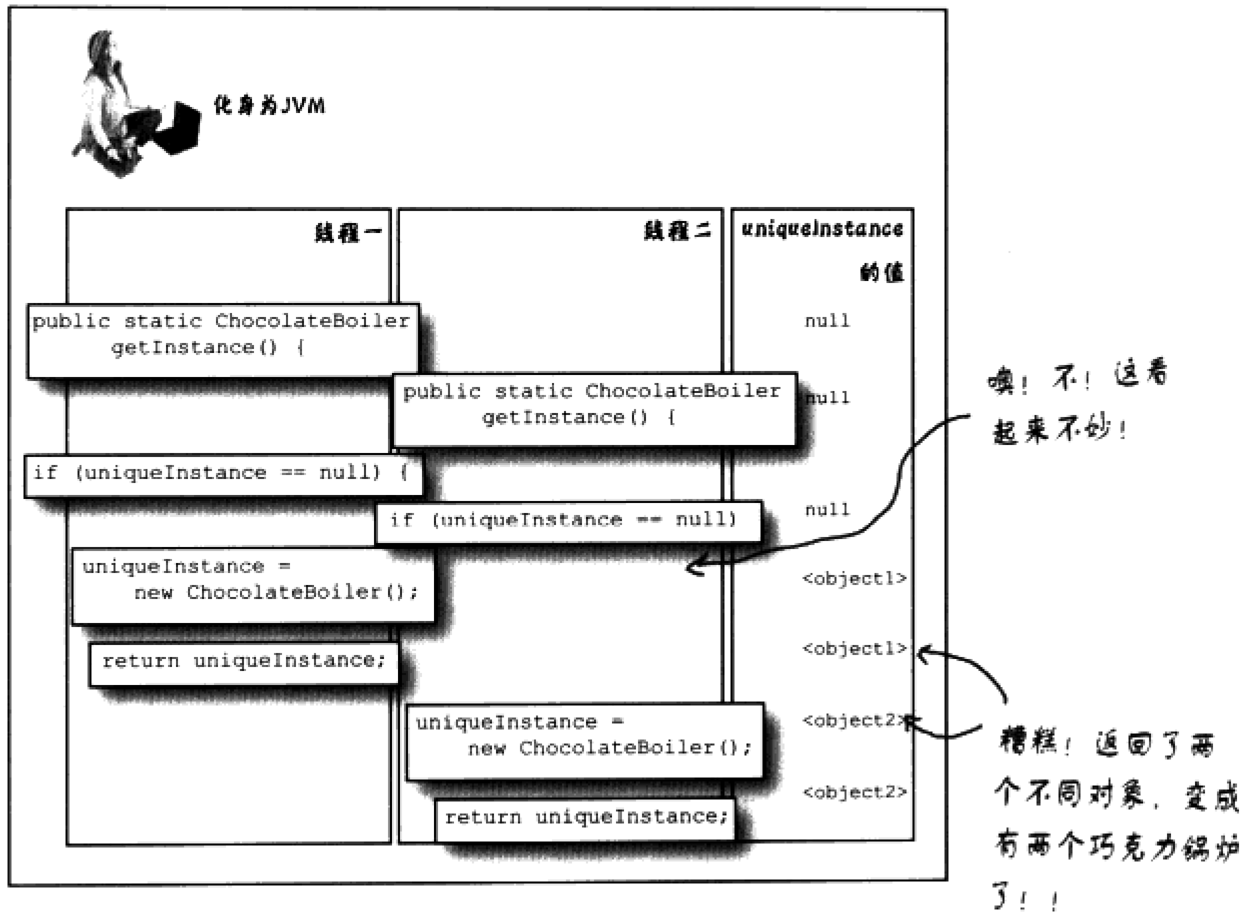
横排提示：

1. It was "one of a kind"
2. Added to chocolate in the boiler
8. An incorrect implementation caused this to overflow
10. Singleton provides a single instance and (three words)
12. Flawed multithreading approach if not using Java 1.5
13. Chocolate capital of the US
14. One advantage over global variables: ________ creation
15. Company that produces boilers
16. To totally defeat the new constructor, we have to declare the constructor ________

竖排提示：

1. Multiple __________ can cause problems
3. A Singleton is a class that manages an instance of ________
4. If you don't need to worry about lazy instantiation, you can create your instance ________
5. Prior to 1.2, this can eat your Singletons (two words)
6. The Singleton was embarassed it had no public ____________
7. The classic implementation doesn't handle this
9. Singleton ensures only one of these exist
11. The Singleton Pattern has one

习题解答



Sharpen your pencil 请帮Choc-O-Holic改进ChocolateBoiler类，把此类设计成单件。

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    public static ChocolateBoiler getInstance() {

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }
    public static ChocolateBoiler getInstance() {
    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            //用牛奶、巧克力混合物填充锅炉
        }
    }
    // 剩余的ChocolateBoiler编码
}
```

习题解答

Sharpen your pencil

描述每一种方案对于修改巧克力锅炉代码所遇到的问题的适用性。

同步getInstance()方法：

这是保证可行的最直接做法。对于巧克力锅炉似乎没有性能的考虑，

所以可以用这个方法。

急切实例化

我们一定需要用到一个巧克力锅炉，所以静态地初始化实例并不是不可以。

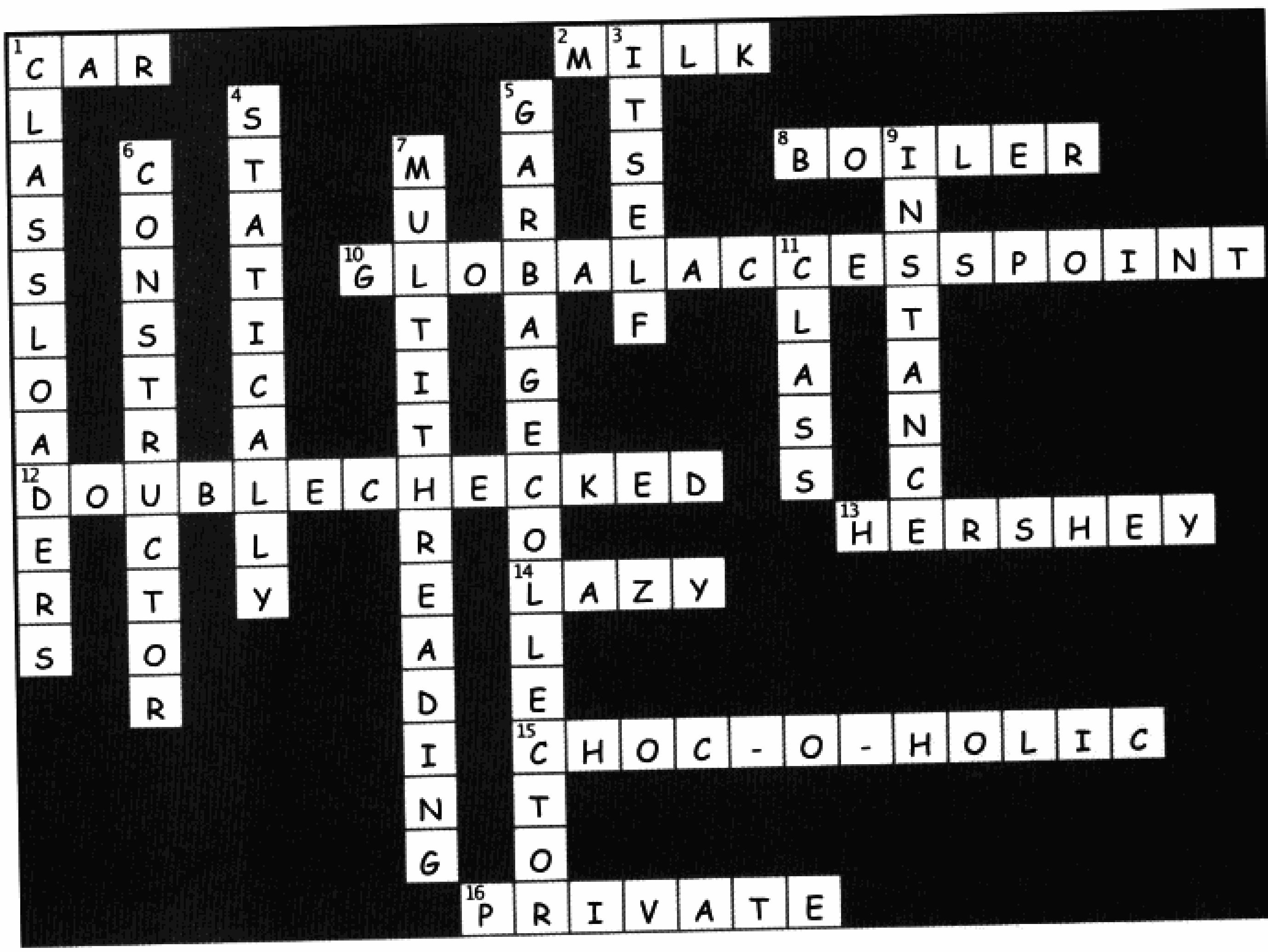
虽然对于采用标准模式的开发人员来说，此做法可能稍微陌生一点。但它还是可行的。

双重检查加锁

由于没有性能上的考虑，所以这个方法似乎杀鸡用了牛刀。另外，采用这个方法

还得确定使用的是Java5以上的版本。

习题解答



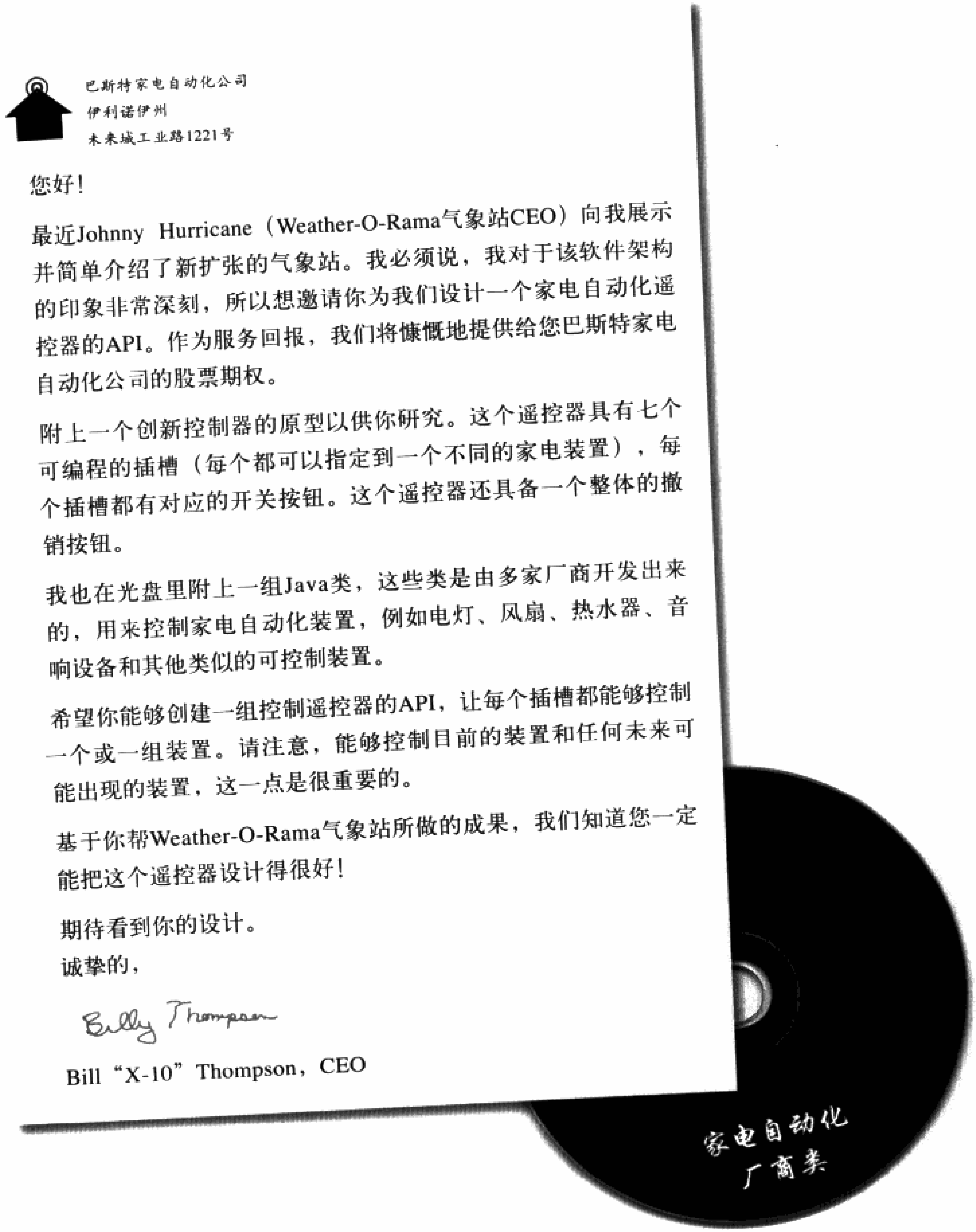
1 CLASSLOADERS
CAR
2 MILK
3 ITSELF
4 STATICALLY
5 GARBAGECOLLECTOR
6 CONSTRUCTOR
7 MULTITHREADING
8 BOILER
9 INSTANCE
10 GLOBALACCESSPOINT
11 CLASS
12 DOUBLECHECKED
13 HERSHEY
14 LAZY
15 CHOC-O-HOLIC
16 PRIVATE

6 命令模式

封装调用



在本章，我们将把封装带到一个全新的境界：把方法调用（method invocation）封装起来。没错，通过封装方法调用，我们可以把运算块包装成形。所以调用此运算的对象不需要关心事情是如何进行的，只要知道如何使用包装成形的方法来完成它就可以。通过封装方法调用，也可以做一些很聪明的事情，例如记录日志，或者重复使用这些封装来实现撤销（undo）。



巴斯特家电自动化公司
伊利诺伊州
未来城工业路1221号

您好！

最近Johnny Hurricane（Weather-O-Rama气象站CEO）向我展示并简单介绍了新扩张的气象站。我必须说，我对于该软件架构的印象非常深刻，所以想邀请你为我们设计一个家电自动化遥控器的API。作为服务回报，我们将慷慨地提供给您巴斯特家电自动化公司的股票期权。

附上一个创新控制器的原型以供你研究。这个遥控器具有七个可编程的插槽（每个都可以指定到一个不同的家电装置），每个插槽都有对应的开关按钮。这个遥控器还具备一个整体的撤销按钮。

我也在光盘里附上一组Java类，这些类是由多家厂商开发出来的，用来控制家电自动化装置，例如电灯、风扇、热水器、音响设备和其他类似的可控制装置。

希望你能够创建一组控制遥控器的API，让每个插槽都能够控制一个或一组装置。请注意，能够控制目前的装置和任何未来可能出现的装置，这一点是很重要的。

基于你帮Weather-O-Rama气象站所做的成果，我们知道您一定能把这个遥控器设计得很好！

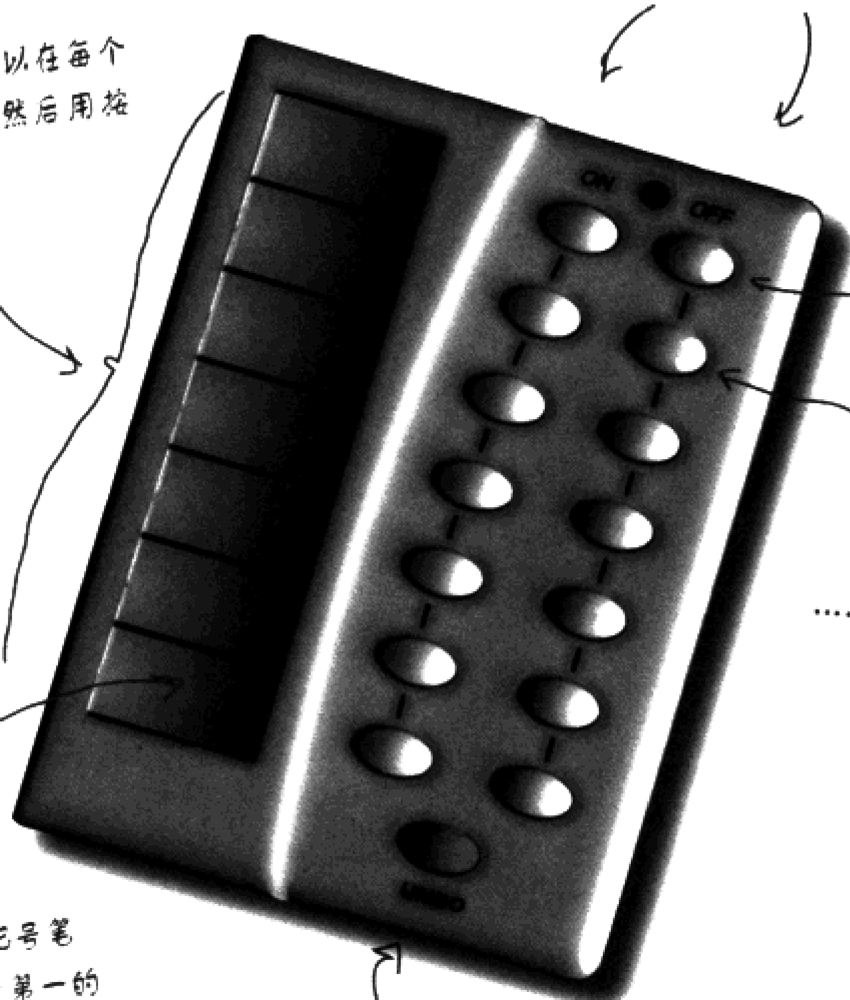
期待看到你的设计。

诚挚的，

Billy Thompson

Bill “X-10” Thompson，CEO

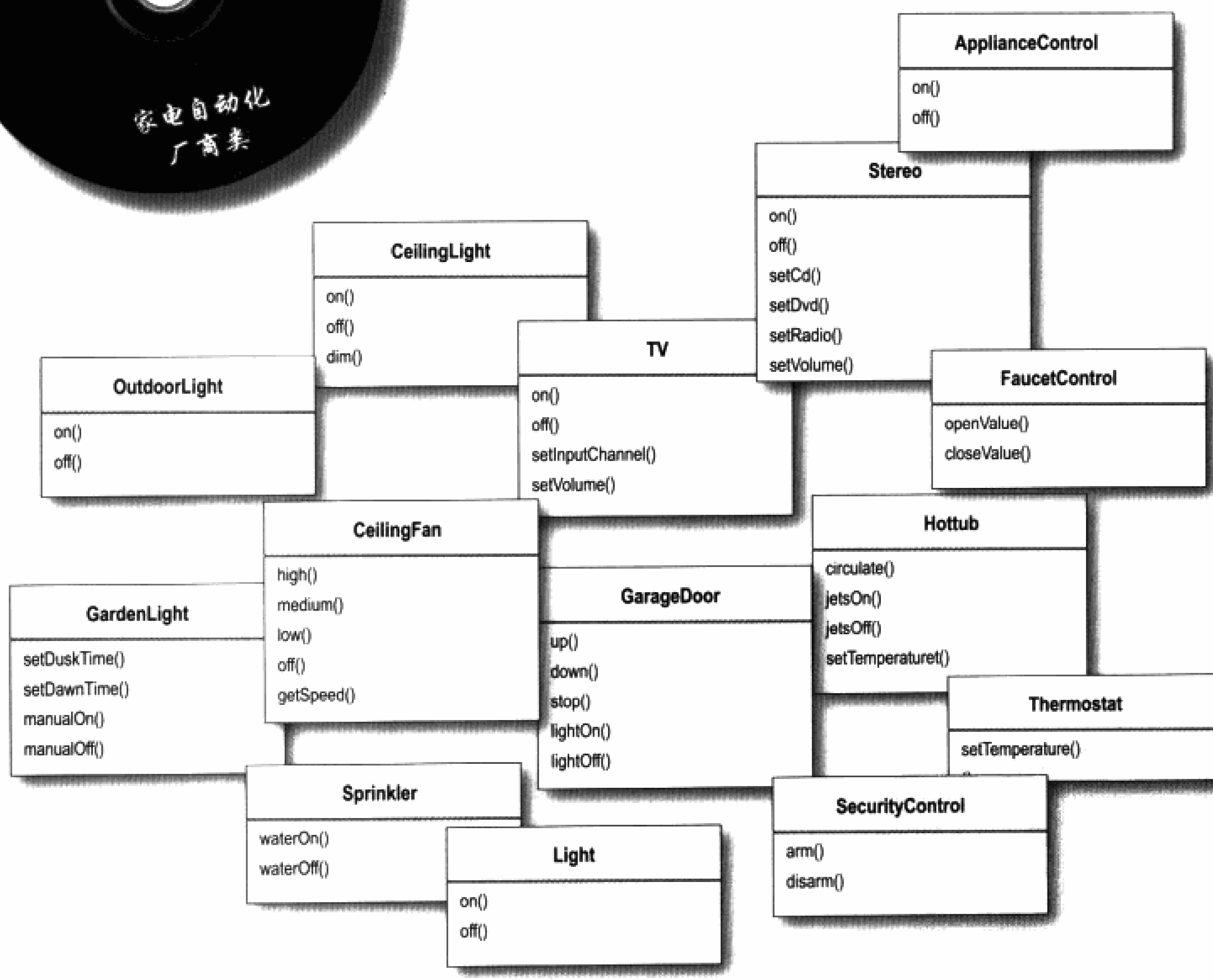
让硬件解脱！让我们看看这个遥控器……





看一下厂商的类

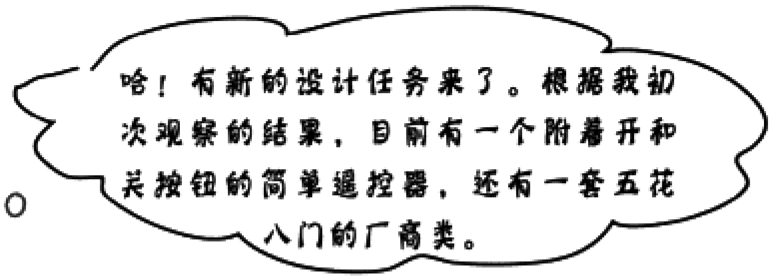
看看光盘上面的厂商类，可以使你对即将设计的对象的接口有一些想法。



看起来类好像不少，但接口各有差异。麻烦还不只是这样，这些类以后还会越来越多。所以设计一个遥控器API变得很有挑战性。让我们继续设计吧！

办公室隔间对话

你的团队正在讨论如何设计这个遥控器API……





Mary：是的，有许多的类都具备on()和off()方法，除此之外，还有一些方法像是dim()、setTemperature()、setVolumn()、setDirection()。

Sue：还不只这样，听起来似乎将来还会有更多的厂商类，而且每个类还会有各式各样的方法。

Mary：我认为要把它看成分离的关注点，这很重要：遥控器应该知道如何解读按钮被按下的动作，然后发出正确的请求，但是遥控器不需知道这些家电自动化的细节，或者如何打开热水器。

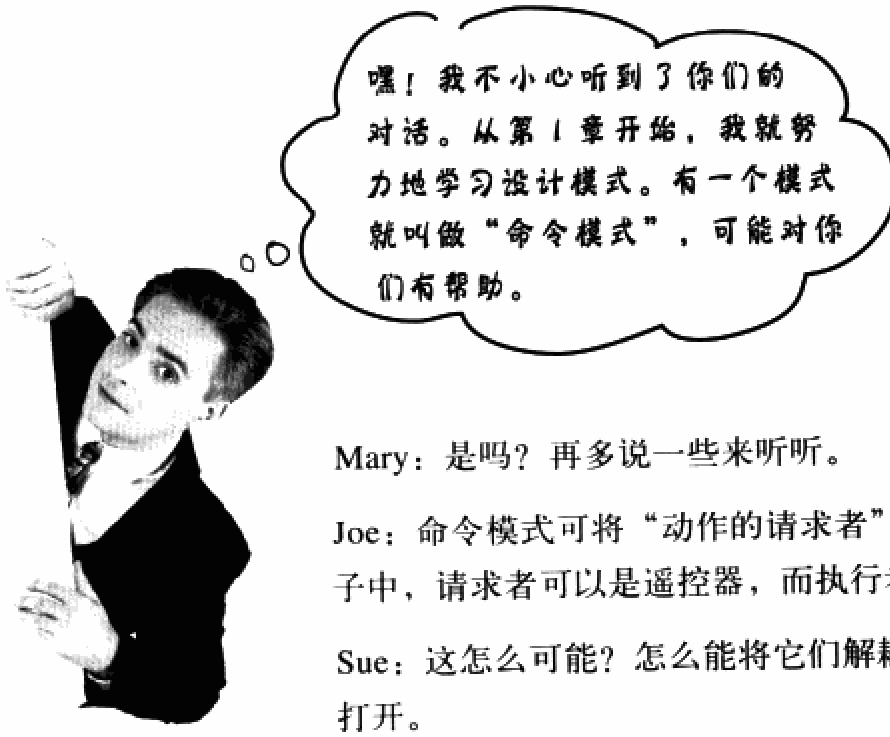
Sue：听起来好像是个不错的设计方式。但如果遥控器很笨，只知道如何做出一般的要求，那又怎能设计出让这个遥控器能够调用一些诸如打开电灯或车库门的动作呢？

Mary：我不确定该怎么做，但是我们不必让遥控器知道太多厂商类的细节。

Sue：你的意思是……

Mary：我们不想让遥控器包含一大堆if语句，例如“if slot1==Light, then light.on(), else if slot1 == Hottub then hottob. jetsOn()”。大家都知道这样的设计很糟糕。

Sue：我同意你的说法。只要有新的厂商类进来，就必须修改代码，这会造成潜在的错误，而且工作没完没了。



Mary：是吗？再多说一些来听听。

Joe：命令模式可将“动作的请求者”从“动作的执行者”对象中解耦。在你们的例子中，请求者可以是遥控器，而执行者对象就是厂商类其中之一的实例。

Sue：这怎么可能？怎么能将它们解耦？毕竟，当我按下按钮时，遥控器必须把电灯打开。

Joe：在你的设计中采用“命令对象”就可以办到。利用命令对象，把请求（例如打开电灯）封装成一个特定对象（例如客厅电灯对象）。所以，如果对每个按钮都存储一个命令对象，那么当按钮被按下的时候，就可以请命令对象做相关的工作。遥控器并不需要知道工作内容是什么，只要有个命令对象能和正确的对象沟通，把事情做好就可以了。所以，看吧，遥控器和电灯对象解耦了。

Sue：的确听起来像是一个正确的方向。

Mary：我仍然无法理解这个模式怎么工作。

Joe：由于对象之间是如此的解耦，要描述这个模式实际的工作并不容易。

Mary：听听我的想法是否正确：使用这个模式，我们能够创建一个API，将这些命令对象加载到按钮插槽，让遥控器的代码尽量保持简单。而把家电自动化的工作和进行该工作的对象一起封装在命令对象中。

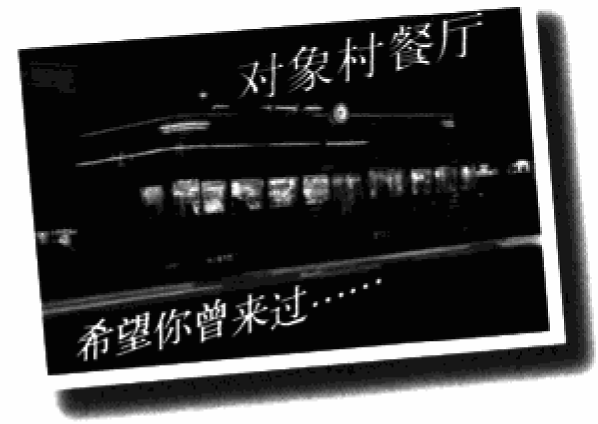
Joe：是的，我也这么认为。我也认为这个模式可以同时帮你设计“撤销按钮”，但我还没研究到这部分。

Mary：听起来令人振奋，但我想应该还要好好学习这个模式。

Sue：我也是。

同时，回到餐厅
……或者该说是
回到命令模式的简单介绍

如同Joe所说的，仅仅通过听别人口述的方式来了解命令模式，确实有点困难。但是别害怕，有一些朋友正准备帮助我们：还记得第1章里出现的友好餐厅吗？距离上次和Alice、Flo及快餐厨师见面已经有好一阵子了。现在我们有很好的理由回去（除了食物和很棒的对话之外）：餐厅可以帮助我们了解命令模式。



所以，让我们再度回到餐厅，研究顾客、女招待、订单，以及快餐厨师之间的交互。通过这样的互动，你将体会到命令模式所涉及的对象，也会知道它们之间如何被解耦。之后，我们就可以解决遥控器API了。

进入对象村餐厅……

我们都知道餐厅是怎么工作的：



❶ 你，也就是顾客，把订单交给女招待。

❷ 女招待拿了订单，放在订单柜台，然后喊了一声“订单来了！”

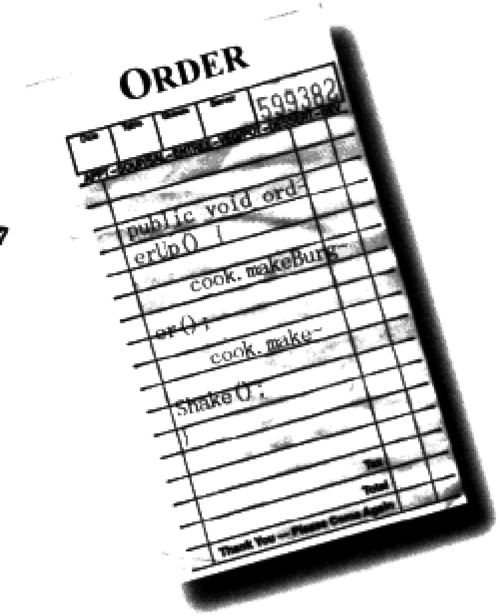
❸ 快餐厨师根据订单准备餐点。

让我们更详细地研究这个交互过程……

……既然餐厅是在对象村，所以让我们也来思考对象和方法的调用关系



对象村餐厅的角色和职责



一张订单封装了准备餐点的请求。

把订单想象成一个用来请求准备餐点的对象，和一般的对象一样，订单对象可以被传递：从女招待传递到订单柜台，或者从女招待传递到接替下一班的女招待。订单的接口只包含一个方法，也就是orderUp()。这个方法封装了准备餐点所需的动作。订单内有一个到“需要进行准备工作的对象”（也就是厨师）的引用。这一切都被封装起来，所以女招待不需要知道订单上有什么，也不需要知道是谁来准备餐点；她只需要将订单放到订单窗口，然后喊一声“订单来了”就可以了。

好吧，在真实的生活里，女招待可能需要关心订单里写些什么和会由谁来准备这一餐，但这里可是对象村……跟着做吧！

女招待的工作是接受订单，然后调用订单的orderUp()方法。

女招待的工作很简单：接下顾客的订单，继续帮助下一个顾客，然后将一定数量的订单放到订单柜台，并调用orderUp()方法，让人来准备餐点。如同在对象村讨论过的，女招待其实不必担心订单的内容是什么，或者由谁来准备餐点。她只需要知道，订单有一个orderUp()方法可以调用，这就够了。

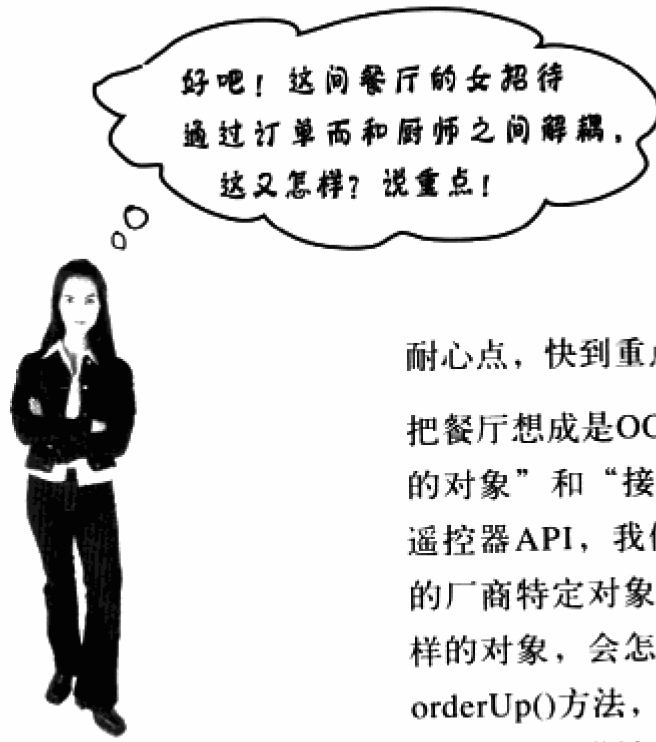
现在，一天内，不同的顾客有不同的订单，这会使得女招待的takeOrder()方法被传入不同的参数。女招待知道所有的订单都支持orderUp()方法，任何时候她需要准备餐点时，调用这个方法就是了。



快餐厨师具备准备餐点的知识。

快餐厨师是一种对象，他真正知道如何准备餐点。一旦女招待调用orderUp()方法，快餐厨师就接手，实现需要创建餐点的所有方法。请注意，女招待和厨师之间是彻底的解耦：女招待的订单封装了餐点的细节，她只要调用每个订单的方法即可，而厨师看了订单就知道该做些什么餐点；厨师和女招待之间从来不需要直接沟通。





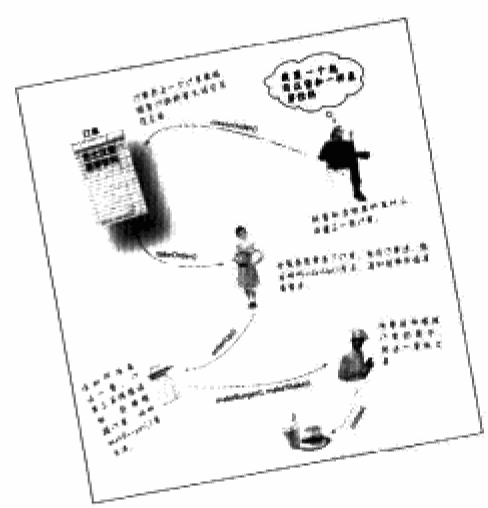
耐心点，快到重点了……

把餐厅想成是OO设计模式的一种模型，而这个模型允许将“发出请求的对象”和“接受与执行这些请求的对象”分隔开来。比方说，对于遥控器API，我们需要分隔开“发出请求的按钮代码”和“执行请求的厂商特定对象”。万一遥控器的每个插槽都持有一个像餐厅订单那样的对象，会怎么样？那么，当一个按钮被按下，只要调用该对象的orderUp()方法，电灯就开了，而遥控器不需要知道事情是怎么发生的，也不需要知道涉及哪些对象。

现在我们就把餐厅的对话换成命令模式……

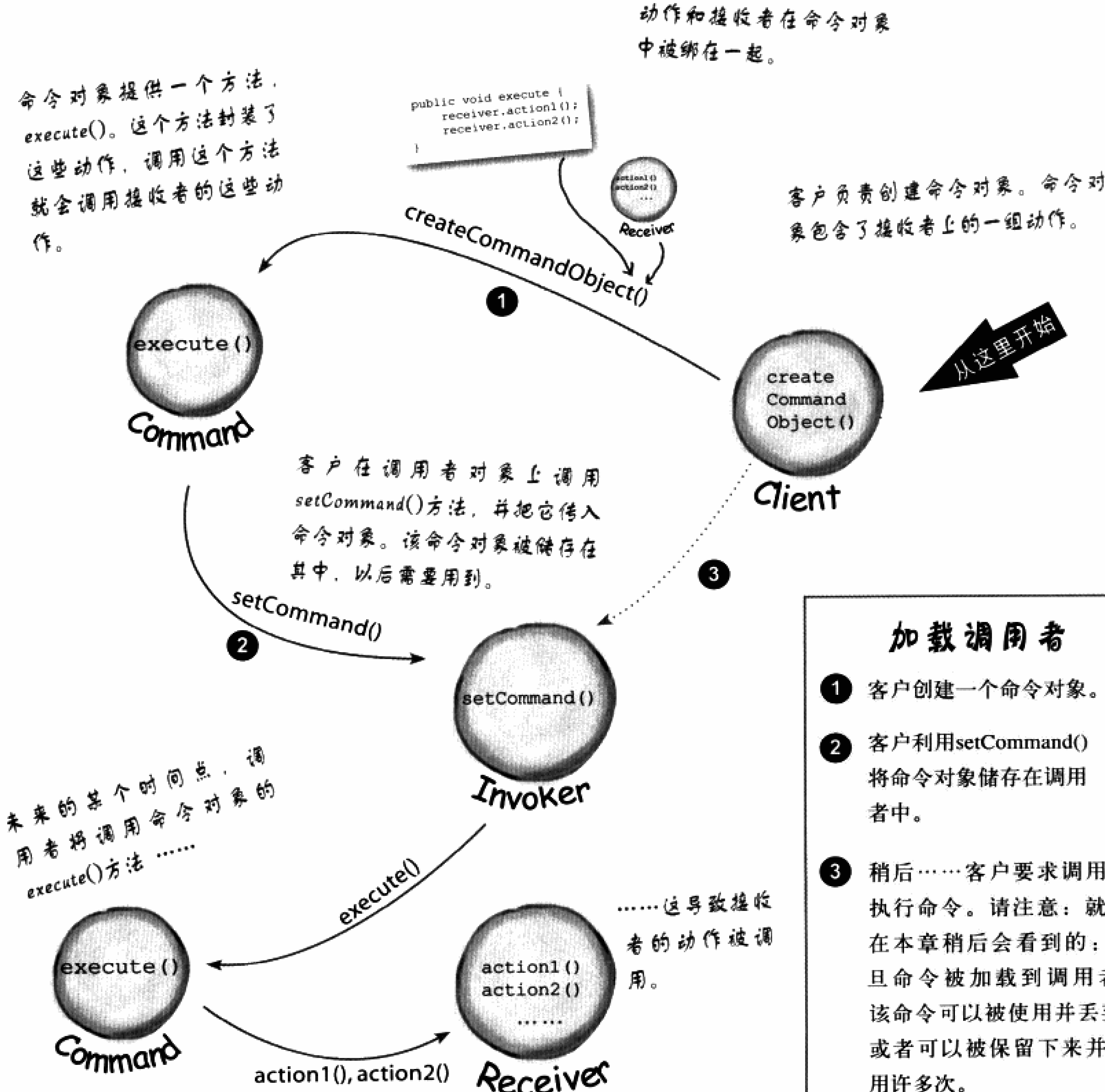
BRAIN POWER

在继续下一页之前，花些时间研究两页前的这张图，图中有餐厅的角色和职责。请务必了解对象村餐厅的对象和他们之间的关系。完成之后，你就可以准备将目光集中在命令模式上了！



从餐厅到命令模式

好了，我们已经花了很多时间在对象村餐厅，也清楚地知道各种角色的特性和他们的职责。现在我们要重新绘制餐厅图以反映出命令模式。所有的角色依然不变，只有名字改变了。



加载调用者

1. 客户创建一个命令对象。
2. 客户利用setCommand()将命令对象储存在调用者中。
3. 稍后……客户要求调用者执行命令。请注意：就像在本章稍后会看到的：一旦命令被加载到调用者，该命令可以被使用并丢弃，或者可以被保留下来并使用许多次。

连连看

请将餐厅的对象和方法对应到命令模式的相应名称。

餐厅	命令模式
女招待	Command
快餐厨师	execute()
orderUp()	Client
订单	Invoker
顾客	Receiver
takeOrder()	setCommand()

第一个命令对象



是我们建立第一个命令对象的时候了！现在开始写一些遥控器的代码。虽然我们还没搞清楚如何设计遥控器的API，但自下而上建造一些东西，可能会有帮助……

实现命令接口

首先，让所有的命令对象实现相同的包含一个方法的接口。在餐厅的例子中，我们称此方法为orderUp()，然而，现在改为一般惯用的名称execute()。

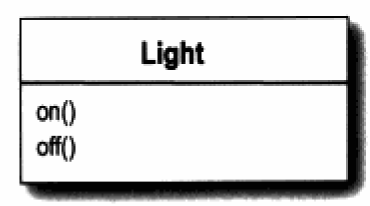
这就是命令接口：

```
public interface Command {
    public void execute();
}
```

简单！只需要一个方法：execute()。

实现一个打开电灯的命令

现在，假设想实现一个打开电灯的命令。根据厂商所提供的类，Light类有两个方法：on()和off()。下面是如何将它实现成一个命令：



这是一个命令，所以需要实现Command接口。

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

构造器被传入了某个电灯（比方说：客厅的电灯），以便让这个命令控制，然后记录在实例变量中。一旦调用execute()，就由这个电灯对象成为接收者，负责接受请求。

这个execute()方法调用接收对象（我们正在控制的电灯）的on()方法。

现在有了LightOnCommad类，让我们看看如何使用它……

使用命令对象

好了，让我们把这一切简化：假设我们有一个遥控器，它只有一个按钮和对应的插槽，可以控制一个装置：

```java
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command) {
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

有一个插槽持有命令，而这个命令控制着一个装置。

这个方法用来设置插槽控制的命令。如果这段代码的客户想要改变遥控器按钮的行为，可以多次调用这个方法。

当按下按钮时，这个方法就会被调用，使得当前命令衔接插槽，并调用它的execute()方法。

遥控器使用的简单测试

下面只有一点点代码，用来测试上面的简单遥控器。我们来看看这个代码，并指出它和命令模式图的对应关系：

这是命令模式的客户。

```java
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

遥控器就是调用者，会传入一个命令对象，可以用来发出请求。

现在创建了一个电灯对象，此对象也就是请求的接收者。

在这里创建一个命令，然后将接收者传给它。

把命令传给调用者。

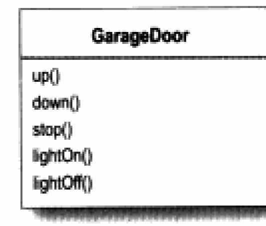
然后模拟按下按钮。

这是执行此测试代码的输出结果！

```
File Edit Window Help DinerFoodYum
%java RemoteControlTest
Light is On
%
```

Sharpen your pencil

好了，现在让你来实现GarageDoorOpenCommand类。先根据GarageDoor类图填好下面的代码。




```
public class GarageDoorOpenCommand
        implements Command {



}
```

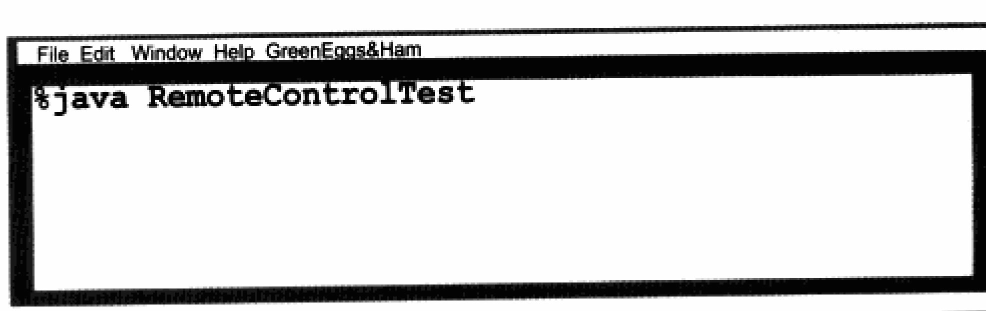
把代码写在这里。

现在你已经有了一个类，下面代码的输出会是什么？（提示：这个GarageDoor的up()方法完成后，将打印出"Garage Door is Open"）。

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);


        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

输出在这里。

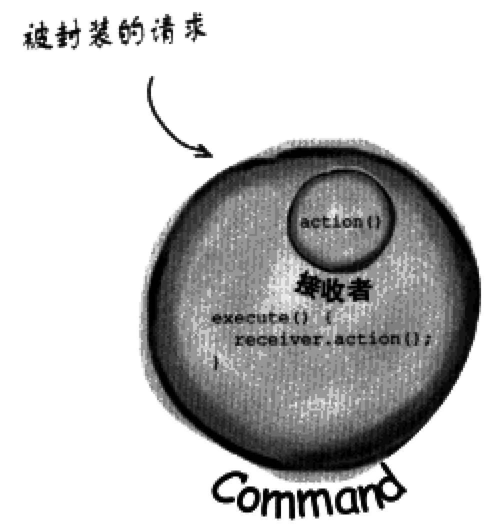


定义命令模式

在经过对象村餐厅的学习之后，你已经实现了部分的遥控器API，而且在这个过程中，你也对命令模式内的类和对象是如何互动的理解得很清楚了。现在，我们就来定义命令模式，并敲定所有的细节。

先从正式的定义开始：

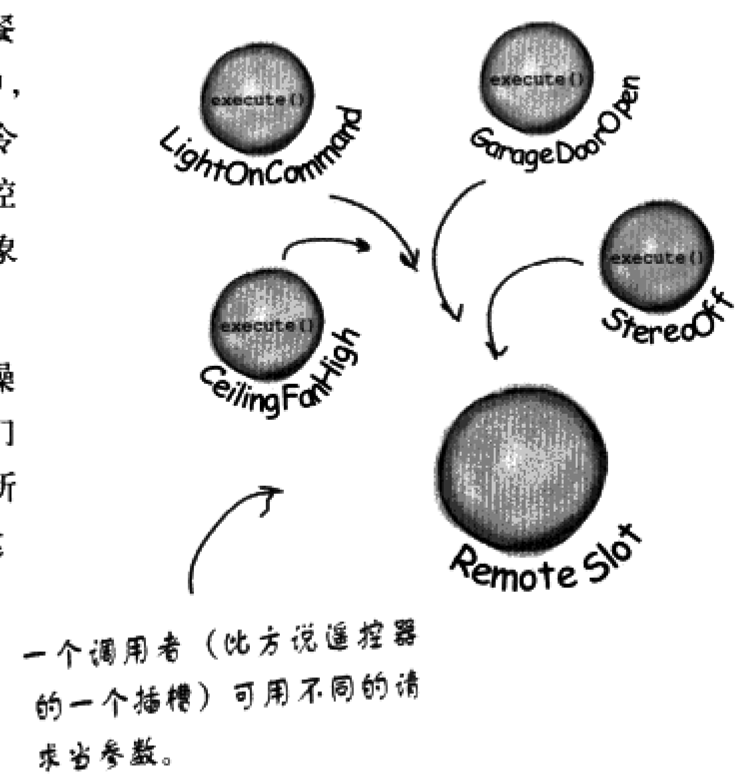
> **命令模式**将“请求”封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。命令模式也支持可撤销的操作。



现在，仔细看这个定义。我们知道一个命令对象通过在特定接收者上绑定一组动作来封装一个请求。要达到这一点，命令对象将动作和接收者包进对象中。这个对象只暴露出一个execute()方法，当此方法被调用的时候，接收者就会进行这些动作。从外面来看，其他对象不知道究竟哪个接收者进行了哪些动作，只知道如果调用execute()方法，请求的目的就能达到。

我们也看到了利用命令来参数化对象的一些例子。再回到餐厅，一整天下来，女招待参数化许多订单。在简单遥控器中，我们先用一个“打开电灯”命令加载按钮插槽，稍后又将命令替换成为另一个“打开车库门”命令。就和女招待一样，遥控器插槽根本不在乎所拥有的是什么命令对象，只要该命令对象实现了Command接口就可以了。

我们还未说到使用命令模式来实现“队列、日志和支持撤销操作”。别担心，这是基本命令模式相当直接的扩展，很快我们就会看到这些内容。一旦有了足够的基础，也可以轻易地持所谓的Meta Command Pattern。Meta Command Pattern 可以创建命令的宏，以便一次执行多个命令。

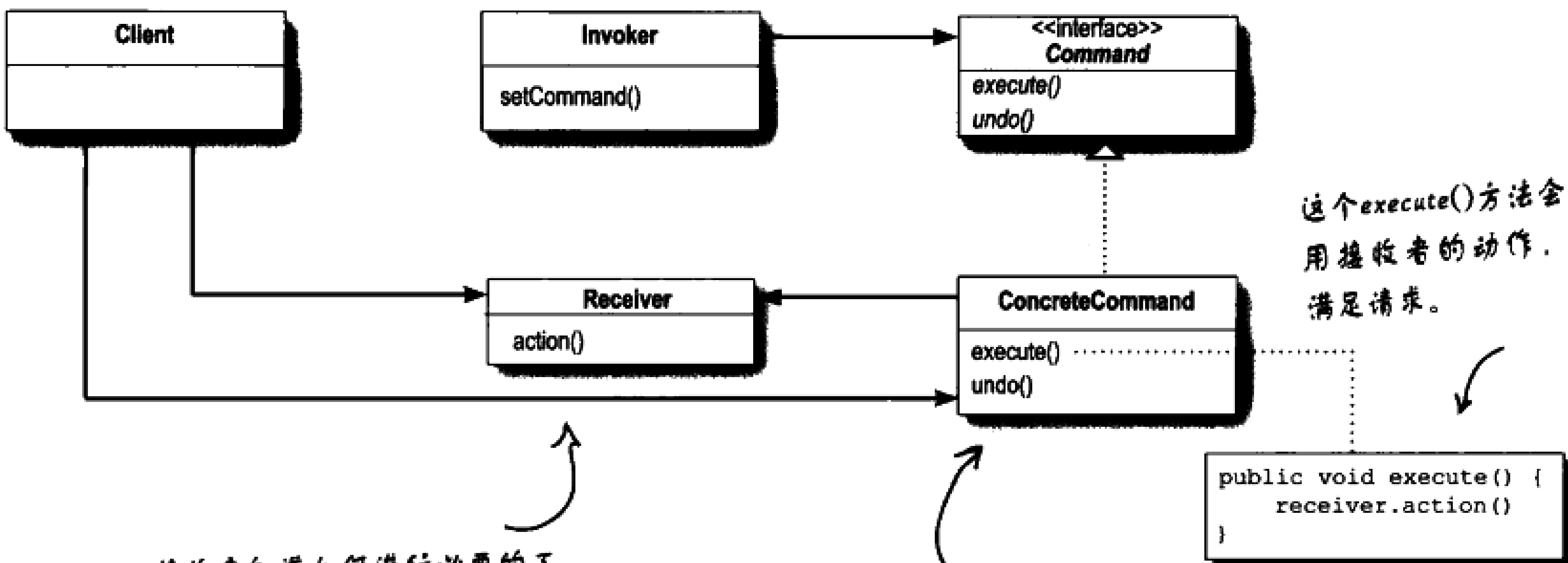


定义命令模式：类图

这个客户负责创建一个ConcreteCommand，并设置其接收者。

这个调用者持有一个命令对象，并在某个时间点调用命令对象的execute()方法，将请求付诸实行。

Command为所有命令声明了一个接口。调用命令对象的execute()方法，就可以让接收者进行相关的动作。这个接口也具备一个undo()方法，本章稍后会介绍这个方法。



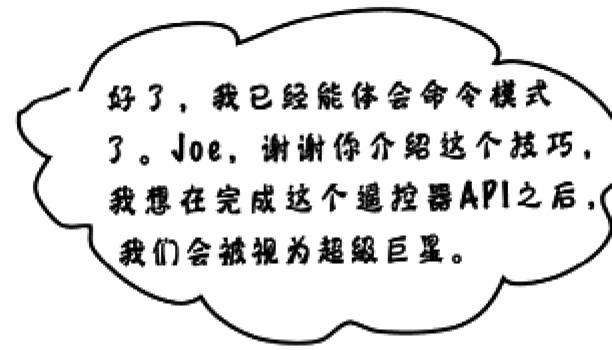
这个execute()方法会调用接收者的动作，以满足请求。

接收者知道如何进行必要的工作，实现这个请求。任何类都可以当接收者。

这个ConcreteCommand定义了动作和接收者之间的绑定关系。调用者只要调用execute()就可以发出请求，然后由ConcreteCommand调用接收者的一个或多个动作。

BRAIN POWER

命令模式的设计如何支持请求调用者和请求接收者之间的解耦？





Mary：我也这么觉得。那么，应该从哪里开始？

Sue：就像我们在简单遥控器（SimpleRemote）中所做的一样，我们需要提供一个方法，将命令指定到插槽。实际上，我们有7个插槽，每个插槽都具备了“开”和“关”按钮，所以我们可以用类似方式，把命令指定给遥控器，像这样：

```
onCommands[0]=onCommand;
offCommands[0]=offCommand;
```

Mary：很有道理，但电灯对象应该排除。遥控器如何分辨客厅或厨房的电灯？

Sue：喔，对了，遥控器无法区分这些！遥控器除了在按下按钮时，调用对应命令对象的execute()方法之外，它什么都不知道。

Mary：是的，这个我似乎了解，但是在实现时，如何确定对象打开（或关闭）正确的装置？

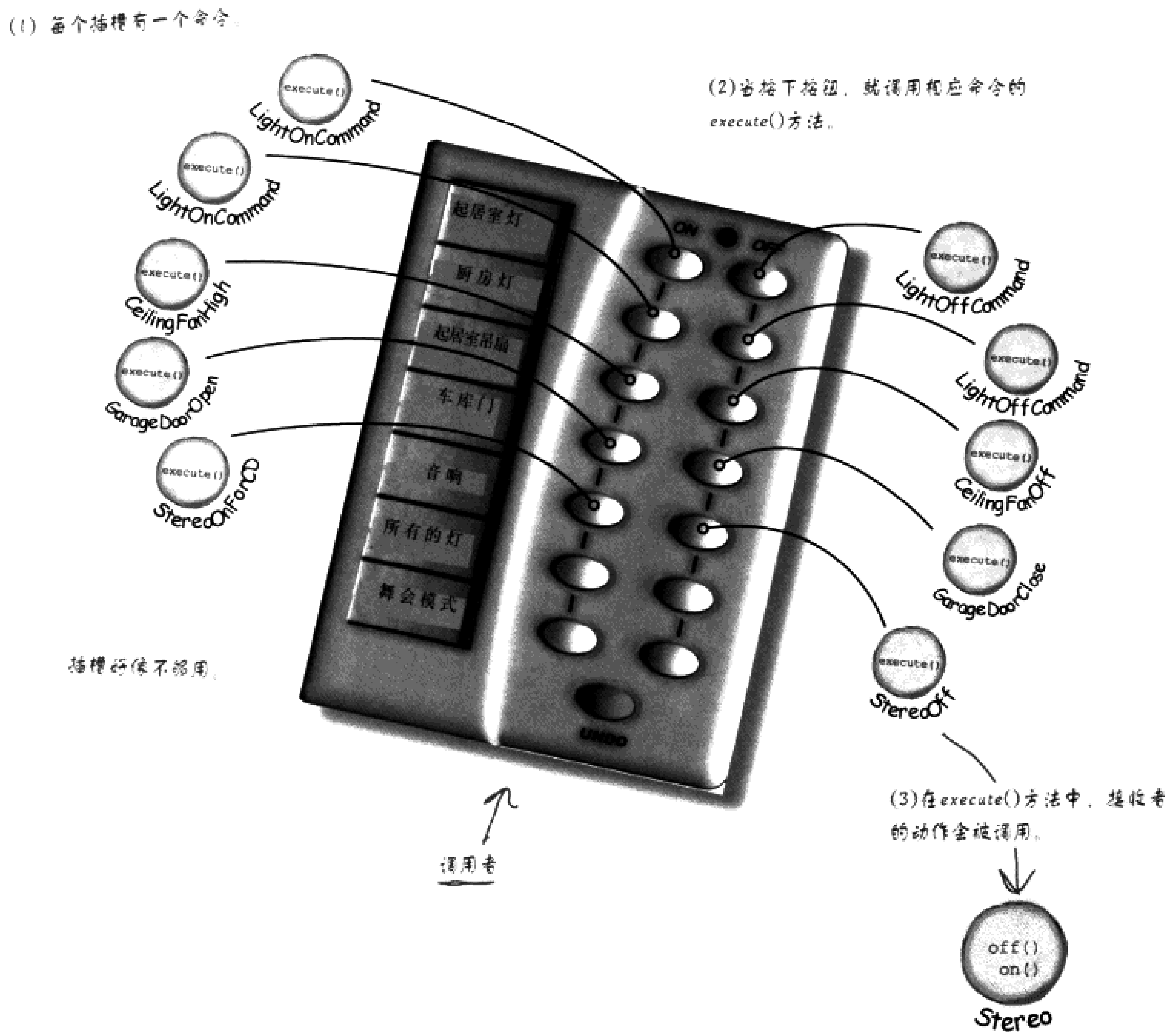
Sue：当我们创建命令并将其加载到遥控器时，我们创建的命令是两个LightCommand，其中一个绑定到客厅电灯对象，另一个则绑定到厨房的电灯对象。别忘了，命令中封装了请求的接收者。所以，在按下按钮时，根本不需要理会打开哪一个电灯，只要execute()被调用，该按钮的对应对象就有动作。

Mary：我想我懂了。现在开始实现这个遥控器吧！我认为一切都会越来越清楚。

Sue：听起来很棒，开工了……

将命令指定到插槽

我们的计划是这样的：我们打算将遥控器的每个插槽，对应到一个命令这样就让遥控器变成“调用者”。当按下按钮，相应命令对象的execute()方法就会被调用，其结果就是，接收者（例如：电灯、天花板电扇、音响）的动作被调用。



实现遥控器

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }


    public String toString() {
        StringBuffer stringBuff = new StringBuffer();
        stringBuff.append("\n------ Remote Control -------\n");
        for (int i = 0; i < onCommands.length; i++) {
            stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
                + "    " + offCommands[i].getClass().getName() + "\n");
        }
        return stringBuff.toString();
    }
}
```

这个时候，遥控器要处理7个开与关的命令，使用相应数组记录这些命令。

在构造器中，只需实例化并初始化这两个开与关的数组。

setCommand()方法须有3个参数，分别是插槽的位置、开的命令、关的命令。这些命令将记录在开关数组中对应的插槽位置，以供稍后使用。

当按下开或关的按钮，硬件就会负责调用对应的方法，也就是onButtonWasPushed()或offButtonWasPushed()。

覆盖toString()，打印出每个插槽和它对应的命令。稍后在测试遥控器的时候，会用到这个方法。

实现命令

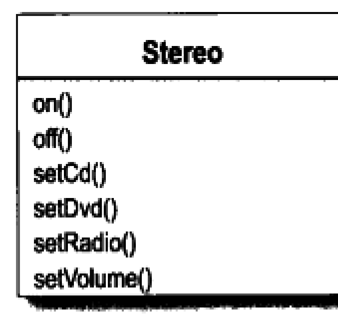
我们已经为SimpleRemoteControl（简单遥控器）动手实现过LightOnCommand，我们可以将相同的代码应用在这里，一切都能顺利进行。关闭命令并没有什么不同，事实上，LightOffCommand看起来就像这样：

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }
}
```

LightOffCommand工作方式和LightOnCommand一样，只是调用不同的方法，也就是off()方法。



让我们来提高挑战性，如何为音响（Stereo）编写开与关的命令？好了，关是很容易，只要把Stereo绑定到StereoOffCommand的off()方法就可以了。开就有点复杂，假设我们要写一个StereoOnWithCDCommand……

```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

就如同LightOnCommand的做法一样，传入音响的实例，然后将其储存在局部实例变量中。

要实现这个请求，需要调用音响的三个方法：首先打开它，然后把它设置成播放CD，最后把音量设置为11。为什么是11？这个嘛，总是比10好些，对吧？

这一切还不错。看看剩下的厂商类，此刻，相信你已经有能力可以完成剩下的命令类了。

逐步测试遥控器

遥控器的工作差不多已经完成，我们剩下要做的事情是运行测试和准备API的说明文档。巴斯特家庭自动化公司一定对我们的成果感到印象深刻，不是吗？我们打算呈现一个绝佳的设计，让他们能够生产易于维护的遥控器。将来，他们也将很容易说服厂商，写一些简单的命令类，因为它们写起来很简单。

开始测试这份代码吧！

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan= new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("");
        Stereo stereo = new Stereo("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn =
                new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff =
                new LightOffCommand(kitchenLight);

        CeilingFanOnCommand ceilingFanOn =
                new CeilingFanOnCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        GarageDoorUpCommand garageDoorUp =
                new GarageDoorUpCommand(garageDoor);
        GarageDoorDownCommand garageDoorDown =
                new GarageDoorDownCommand(garageDoor);

        StereoOnWithCDCommand stereoOnWithCD =
                new StereoOnWithCDCommand(stereo);
        StereoOffCommand  stereoOff =
                new StereoOffCommand(stereo);
```

将所有的装置创建在合适的位置。

创建所有的电灯命令对象。

创建吊扇的开与关命令。

创建车库门的上与下命令。

创建音响的开与关命令。

```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

现在已经有了全部的命令，可以将它们加载到遥控器插槽中。

在这里，使用toString()方法，打印出每个遥控器的插槽和它被指定的命令。

好了，一切准备就绪！现在，逐步按下每个插槽的开与关按钮。

现在，看看遥控器的测试结果……

File Edit Window Help CommandsGetThingsDone

```
% java RemoteLoader
------ Remote Control -------
[slot 0] headfirst.command.remote.LightOnCommand        headfirst.command.remote.LightOffCommand
[slot 1] headfirst.command.remote.LightOnCommand        headfirst.command.remote.LightOffCommand
[slot 2] headfirst.command.remote.CeilingFanOnCommand   headfirst.command.remote.CeilingFanOffCommand
[slot 3] headfirst.command.remote.StereoOnWithCDCommand headfirst.command.remote.StereoOffCommand
[slot 4] headfirst.command.remote.NoCommand             headfirst.command.remote.NoCommand
[slot 5] headfirst.command.remote.NoCommand             headfirst.command.remote.NoCommand
[slot 6] headfirst.command.remote.NoCommand             headfirst.command.remote.NoCommand

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off
%
```

开启插槽　关闭插槽

命令的执行结果！记住，每个装置的输出都是由厂商类所提供的。比方说，当一个电灯对象被打开，它会打印出"Living Room light is on."。



被你发现了。我们的确省略了一些东西。在遥控器中，我们不想每次都检查是否某个插槽都加载了命令。比方说，在这个onButtonWasPushed()方法中，我们可能需要这样的代码：

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

所以，我们要如何避免上述的做法？实现一个不做事情的命令！

```
public class NoCommand implements Command {
    public void execute() { }
}
```

这么一来，在RemoteControl构造器中，我们将每个插槽都预先指定成NoCommand对象，以便确定每个插槽永远都有命令对象。

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

所以在测试的输出中，没有被明确指定命令的插槽，其命令将是默认的NoCommand对象。

模式荣誉奖

NoCommand对象是一个空对象（null object）的例子。当你不想返回一个有意义的对象时，空对象就很有用。客户也可以将处理null的责任转移给空对象。举例来说，遥控器不可能一出厂就设置了有意义的命令对象，所以提供了NoCommand对象作为代用品，当调用它的execute()方法时，这种对象什么事情都不做。

在许多设计模式中，都会看到空对象的使用。甚至有些时候，空对象本身也被视为是一种设计模式。

写文档的时刻到了……

为巴斯特家电自动化公司设计的遥控器API。

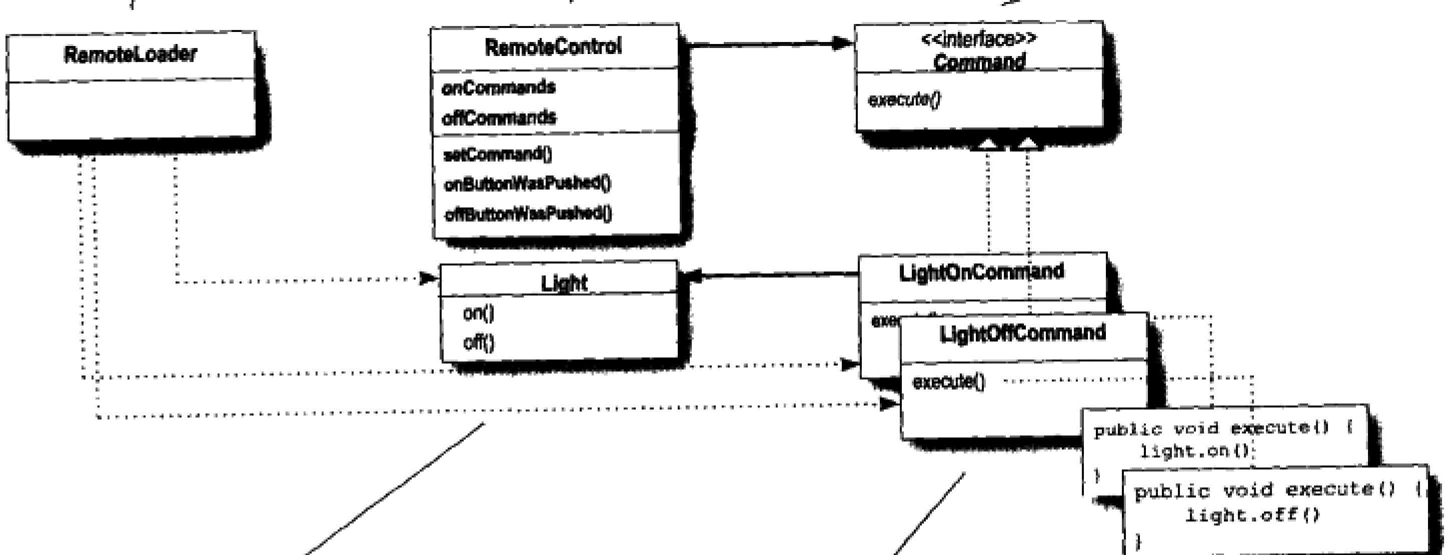
我们很高兴为您呈献下列的家电自动化遥控器设计与应用编程接口。主要的设计目标是让遥控器代码尽可能地简单，这样一来，新的厂商类一旦出现，遥控器并不需要随之修改。因此，我们采用了命令模式，从逻辑上将遥控器的类和厂商的类解耦。我们相信这将降低遥控器的生产成本，并大大地减少未来维护时所需的费用。

下面的类图提供了设计的全貌：

RemoteControl管理一组命令对象，每个按钮都有一个命令对象。每当按下按钮，就调用相应的xxButtonWasPushed()方法，间接造成该命令的execute()方法被调用。

所有的遥控器命令都实现这个Command接口，此接口中包含了一个方法，也就是execute()。命令封装了某个特定厂商类的一组动作，遥控器可以通过调用execute()方法，执行这些动作。

RemoteLoader创建许多命令对象，然后将其加载到遥控器的插槽中。每个命令对象都封装了某个家电自动化装置的一项请求。



这些厂商类被用来控制特定的家电自动化装置。在这里，我们用Light类当做例子。

利用Command接口，每个动作都被实现成一个简单的命令对象。命令对象持有对一个厂商类的实例的引用，并实现了一个execute()方法。这个方法会调用厂商类实例的一个或多个方法，完成特定的行为。在这个例子中，有两个类，分别打开电灯与关闭电灯。



哎呀！差点就忘了……还好，因为我们采用基本命令类，所以可以很容易地加上撤销的功能。让我们逐步为遥控器加上撤销命令……

我们要做什么？

好了，我们现在需要在遥控器上加上撤销的功能。这个功能使用起来就像是这样的：比方说客厅的电灯是关闭的，然后你按下遥控器上的开启按钮，自然电灯就被打开了。现在如果按下撤销按钮，那么上一个动作将被倒转，在这个例子里，电灯将被关闭。在进入更复杂的例子之前，先让撤销按钮能够处理电灯：

1. 当命令支持撤销时，该命令就必须提供和execute()方法相反的undo()方法。不管execute()刚才做什么，undo()都会倒转过来。这么一来，在各个命令中加入undo()之前，我们必须先在Command 接口中加入undo()方法：

```
public interface Command {
    public void execute();
    public void undo();
}
```

这是新加入的undo()方法。

这实在是够简单。

现在让我们深入电灯的命令，并实现undo()方法。

❷ 我们从LightOnCommand开始下手：如果LightOnCommand的execute()方法被调用，那么最后被调用的是on()方法。我们知道undo()需要调用off()方法进行相反的动作。

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

execute()打开电灯，所以undo()该做的事情就是关闭电灯。

太容易了！现在来处理LightOffCommand，在这里，undo()方法需要调用电灯的on()方法。

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}
```

在这里，undo()把电灯打开！

实在是简单到不行！事情可还没完，我们还要花一些力气，让遥控器能够追踪最后被按下的按钮是什么。

❸ 要加上对撤销按钮的支持，我们必须对遥控器类做一些小修改。我们打算这么做：加入一个新的实例变量，用来追踪最后被调用的命令，然后，不管何时撤销按钮被按下，我们都可以取出这个命令并调用它的undo()方法。

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public RemoteControlWithUndo() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }

    public void undoButtonWasPushed() {
        undoCommand.undo();
    }

    public String toString() {
        // 这里是toString代码……    }
}
```

前一个命令将被记录在这里。

一开始，并没有所谓的“前一个命令”，所以将它设置成NoCommand的对象。

当按下按钮，我们取得这个命令，并优先执行它，然后将它记录在undoCommand实例变量中。不管是“开”或“关”命令，我们的处理方法都是一样的。

当按下撤销按钮，我们调用undoCommand实例变量的undo()方法，就可以倒转前一个命令。

QA时间

好了！让我们修改测试程序，测试撤销按钮。

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Light livingRoomLight = new Light("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

- 创建一个电灯对象和新支持undo()功能的命令。
- 将电灯命令设置到遥控器的0号插槽。
- 打开电灯，关闭电灯，然后撤销。
- 关闭电灯，打开电灯，然后撤销。

结果如下：

```
File Edit Window Help UndoCommandsDefyEntropy
% java RemoteLoader
Light is on
Light is off

------ Remote Control -------
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOffCommand

Light is on

Light is off
Light is on

------ Remote Control -------
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOnCommand

Light is off
```

- 打开电灯，关闭电灯。
- 这是电灯的命令。
- 现在undo变量记录的是LightOffCommand。
- 按下撤销按钮……LightOffCommand的undo()会造成电灯被打开。
- 关闭电灯，再打开电灯
- 现在undo变量记录的是LightOnCommand。
- 按下撤销按钮，造成电灯被关闭。

使用状态实现撤销

好了，实现电灯的撤销是有意义的，但也实在是太容易了。通常，想要实现撤销的功能，需要记录一些状态。让我们试一个更有趣的例子，比方说厂商类中的天花板上的吊扇。吊扇允许有多种转动速度，当然也允许被关闭。

CeilingFan
high() medium() low() off() getSpeed()

吊扇的源码如下：

```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;
    String location;
    int speed;

    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        // 设置高转速
    }

    public void medium() {
        speed = MEDIUM;
       // 设置中转速
    }

    public void low() {
        speed = LOW;
        // 设置低转速
    }

    public void off() {
        speed = OFF;
        // 关闭吊扇
    }

    public int getSpeed() {
        return speed;
    }
}
```

注意，吊扇的类中具有局部状态，代表吊扇的速度。

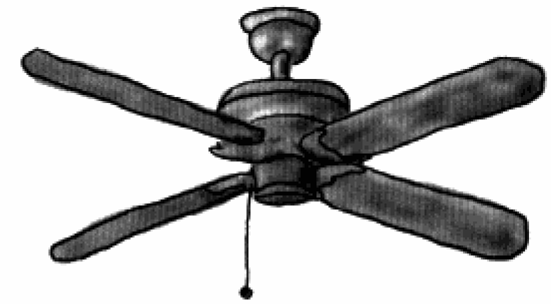
嗯，想要正确地实现undo，就必须把吊扇以前的速度考虑进去。



这些方法用来设置吊扇速度。

可以利用getSpeed()方法得到吊扇的当前速度。

加入撤销到吊扇的命令类



现在就让我们把撤销加入天花板吊扇的诸多命令中。这么做，需要追踪吊扇的最后设置速度，如果undo()方法被调用了，就要恢复成之前吊扇速度的设置值。下面是CeilingFanHighCommand的代码：

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

增加局部状态以便追踪吊扇之前的速度。

在execute()中，在我们改变吊扇的速度之前，需要先将它之前的状态记录起来，以便需要撤销时使用。

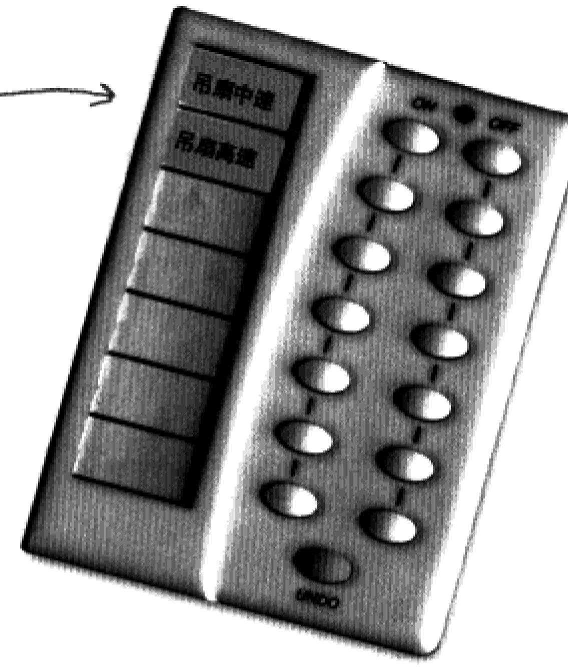
将吊扇的速度设置回之前的值，达到撤销的目的。

BRAIN POWER

我们还有三个天花板吊扇的命令要写：low（低速）、medium（中速）、off（关闭）。你知道如何实现这些命令吗？

准备测试天花板吊扇

该是测试天花板吊扇的时候了。我们打算把第0号插槽的开启按钮设置为中速，把第1号插槽的开启按钮设置为高速，而两个对应的关闭按钮，都是关闭吊扇的命令。



测试脚本如下：

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        CeilingFan ceilingFan = new CeilingFan("Living Room");

        CeilingFanMediumCommand ceilingFanMedium =
                new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
                new CeilingFanHighCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();

        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

在这里实例化了三个命令，分别是：高速、中速和关闭。

在这里将中速设置到第0号插槽，将高速设置到第1号插槽，并加载这两个插槽的关闭命令。

首先，以中速开启吊扇。

然后关闭。

撤销！应该会回到中速……

这个时候开启高速。

再进行一次撤销，应该会回到中速。

测试天花板吊扇

好了，拿起遥控器，加载这些命令，然后按一些按钮！

File Edit Window Help UndoThis!

```
% java RemoteLoader

Living Room ceiling fan is on medium
Living Room ceiling fan is off

------ Remote Control -------
[slot 0] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 1] headfirst.command.undo.CeilingFanMediumCommand    headfirst.command.undo.CeilingFanOff-
Command
[slot 2] headfirst.command.undo.CeilingFanHighCommand    headfirst.command.undo.CeilingFanOffCom-
mand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanOffCommand

Living Room ceiling fan is on medium
Living Room ceiling fan is on high

------ Remote Control -------
[slot 0] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 1] headfirst.command.undo.CeilingFanMediumCommand    headfirst.command.undo.CeilingFanOff-
Command
[slot 2] headfirst.command.undo.CeilingFanHighCommand    headfirst.command.undo.CeilingFanOffCom-
mand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanHighCommand

Living Room ceiling fan is on medium

%
```

以中速打开天花板吊扇，然后将其关闭。

这是遥控器的命令……

……而undo记录了最后执行的命令，也就是CeilingFanOffCommand。

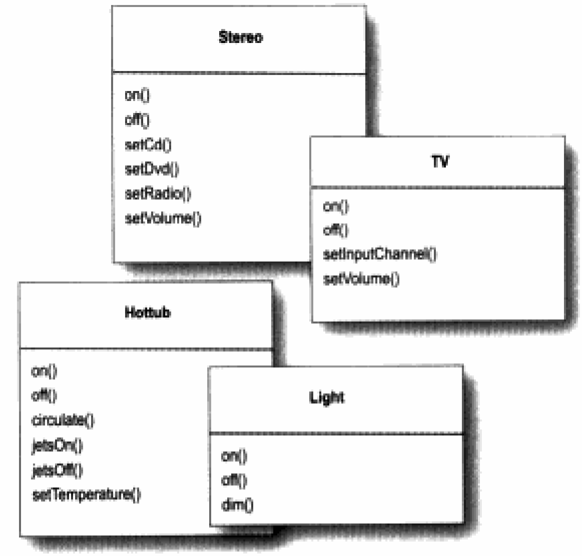
撤销最后一个命令，回到了中速。

现在，转到高速。

现在，高速是最后被执行的命令。

再一次撤销，天花板吊扇回到中速。

每个遥控器都需具备“Party模式”！



如果拥有了一个遥控器，却无法光凭按下一个按钮，就同时能弄暗灯光、打开音响和电视、设置好DVD，并让热水器开始加温，那么要这个遥控器还有什么意义？





Mary的想法是，制造一种新的命令，用来执行其他一堆命令……而不只是执行一个命令！这个想法不错吧？

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
```

在宏命令中，用命令数组存储一大堆命令。

当这个宏命令被遥控器执行时，就一次性执行数组里的每个命令。

使用宏命令

让我们逐步来看如何使用宏命令：

1. 先创建想要进入宏的命令集合：

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();

LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

创建所有的装置，电灯、电视、音响和热水器。

现在，创建所有的On命令来控制它们。

Sharpen your pencil 我们也需要关闭按钮的命令，请在这里写下创建它们的代码：

2. 接下来创建两个数组，其中一个用来记录开启命令，另一个用来记录关闭命令，并在数组内放入对应的命令：

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};

MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

一个数组用来记录开启命令，另一个数组用来记录关闭命令……

……然后创建两个对应的宏持有它们。

3. 然后将宏命令指定给我们所希望的按钮：

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

将宏命令指定给一个按钮。

❹ 最后，只需按下一些按钮，测试是否正常工作。

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

输出如下：

File Edit Window Help You Can'tBeatABabka

```
% java RemoteLoader
------ Remote Control -------
[slot 0] headfirst.command.party.MacroCommand    headfirst.command.party.MacroCommand
[slot 1] headfirst.command.party.NoCommand    headfirst.command.party.NoCommand
[slot 2] headfirst.command.party.NoCommand    headfirst.command.party.NoCommand
[slot 3] headfirst.command.party.NoCommand    headfirst.command.party.NoCommand
[slot 4] headfirst.command.party.NoCommand    headfirst.command.party.NoCommand
[slot 5] headfirst.command.party.NoCommand    headfirst.command.party.NoCommand
[slot 6] headfirst.command.party.NoCommand    headfirst.command.party.NoCommand
[undo] headfirst.command.party.NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees

%
```

这是两个宏命令。

当我们调用开启宏时，此宏内所有的命令都被执行了……

当我们调用关闭宏时，也没有问题。

练习

我们的宏命令唯一缺少的是撤销功能。一个宏命令被执行完，然后按下撤销按钮，那么宏内所进行的每一道命令都必须被撤销。请在下面的代码中，填入undo()方法的内容：

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {



    }
}
```

there are no Dumb Questions

问： 接收者一定有必要存在吗？为何命令对象不直接实现execute()方法的细节？

答： 一般来说，我们尽量设计"傻瓜"命令对象，它只懂得调用一个接收者的一个行为。然而，有许多"聪明"命令对象会实现许多逻辑，直接完成一个请求。当然你可以设计聪明的命令对象，只是这样一来，调用者和接收者之间的解耦程度是比不上"傻瓜"命令对象的，而且，你也不能够把接收者当做参数传给命令。

问： 我如何能够实现多层次的撤销操作？换句话说，我希望能够按下撤销按钮许多次，撤销到很早很早以前的状态。

答： 好问题！其实这相当容易做到，不要只是记录最后一个被执行的命令，而使用一个堆栈记录操作过程的每一个命令。然后，不管什么时候按下了撤销按钮，你都可以从堆栈中取出最上层的命令，然后调用它的undo()方法。

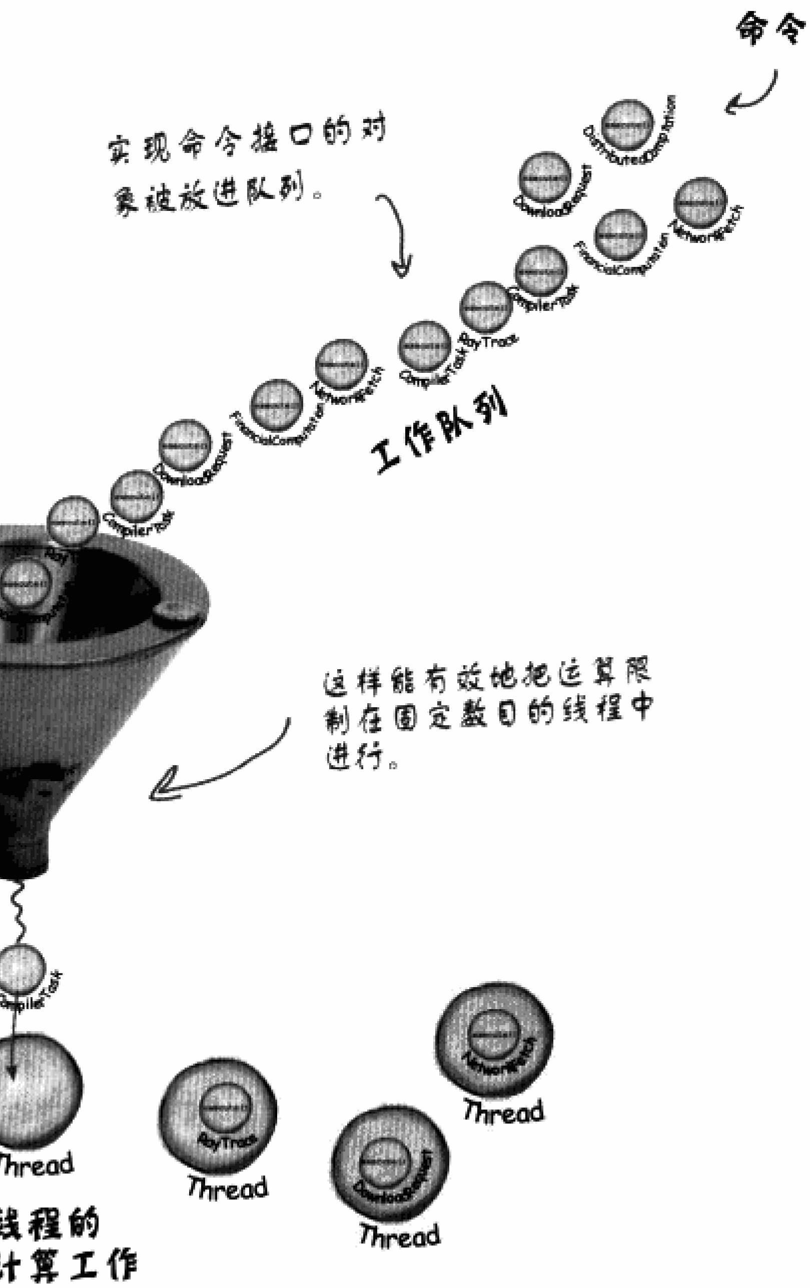
问： 我可以创建一PartyCommand，然后在它的execute()方法中调其他的命令，利用这种做法实现Party模式（Party Mode）吗？

答： 你可以这么做。然而，这等于把Party模式"硬编码"到PartyCommand中。为什么要这么麻烦呢？利用宏命令，你可以动态地决定PartyCommand是由哪些命令组成，所以宏命令在使用上更灵活。一般来说，宏命令的做法更优雅，也需要较少的新代码。

命令模式的更多用途：队列请求

命令可以将运算块打包（一个接收者和一组动作），然后将它传来传去，就像是一般的对象一样。现在，即使在命令对象被创建许久之后，运算依然可以被调用。事实上，它甚至可以在不同的线程中被调用。我们可以利用这样的特性衍生一些应用，例如：日程安排（Scheduler）、线程池、工作队列等。

想象有一个工作队列：你在某一端添加命令，然后另一端则是线程。线程进行下面的动作：从队列中取出一个命令，调用它的execute()方法，等待这个调用完成，然后将此命令对象丢弃，再取出下一个命令……



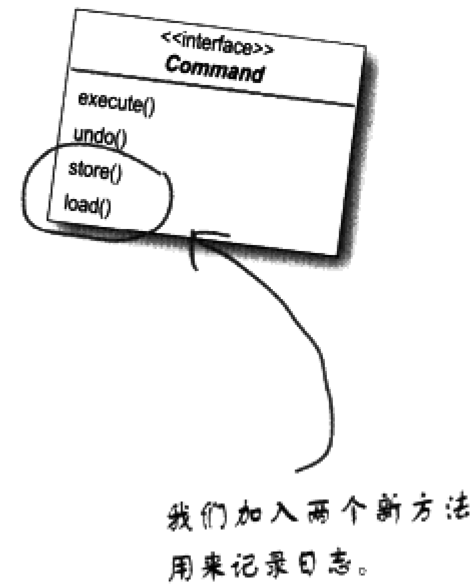
请注意，工作队列类和进行计算的对象之间完全是解耦的。此刻线程可能在进行财务运算，下一刻却在读取网络数据。工作队列对象不在乎到底做些什么，它们只知道取出命令对象，然后调用其execute()方法。类似地，它们只要是实现命令模式的对象，就可以放入队列里，当线程可用时，就调用此对象的execute()方法。

BRAIN POWER

你认为Web服务器如何应用这样的队列方式？还能想到任何其他的应用吗？

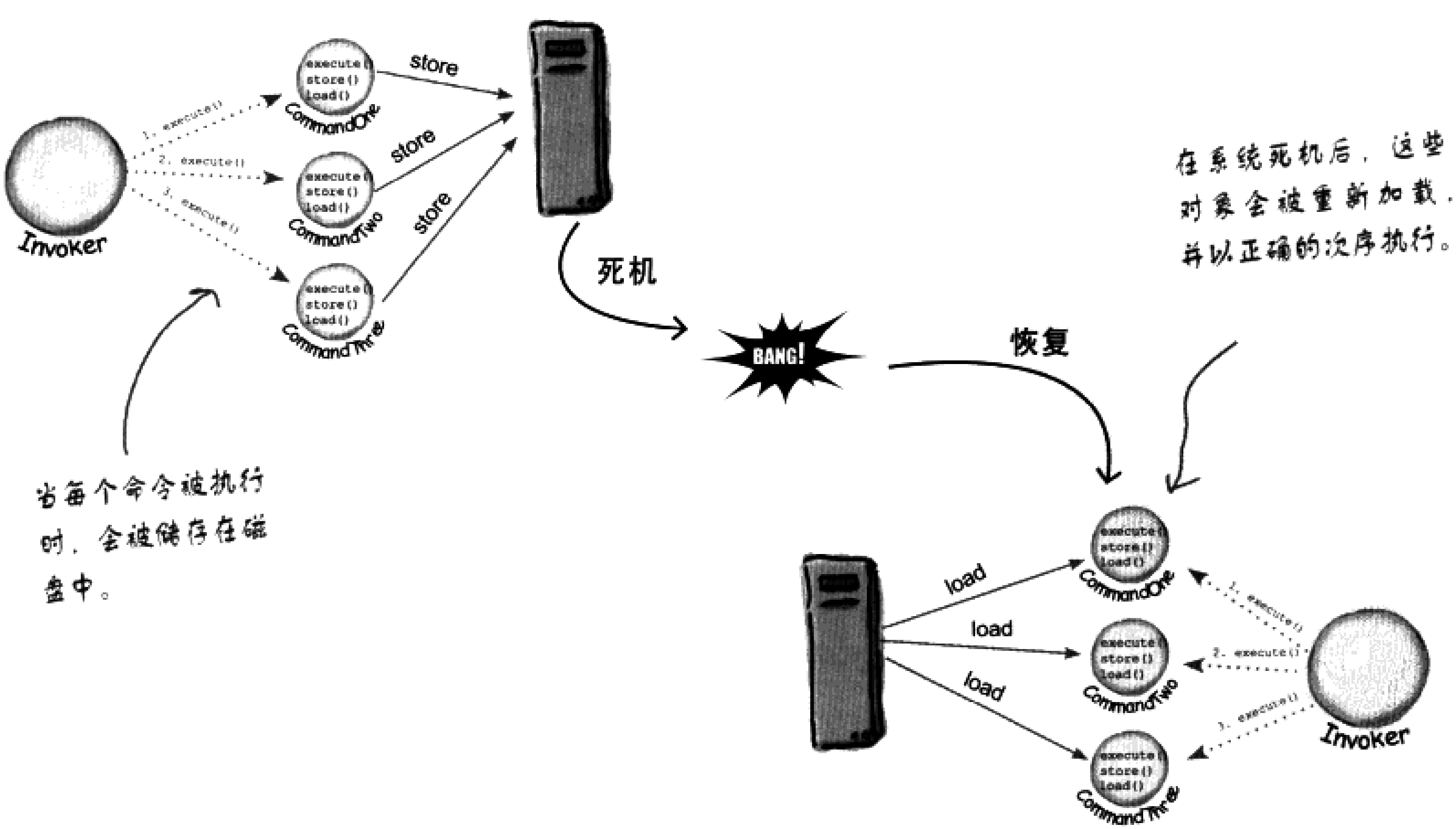
命令模式的更多用途：日志请求

某些应用需要我们将所有的动作都记录在日志中，并能在系统死机之后，重新调用这些动作恢复到之前的状态。通过新增两个方法（store()、load()），命令模式就能够支持这一点。在Java中，我们可以利用对象的序列化（Serialization）实现这些方法，但是一般认为序列化最好还是只用在对象的持久化上（persistence）。



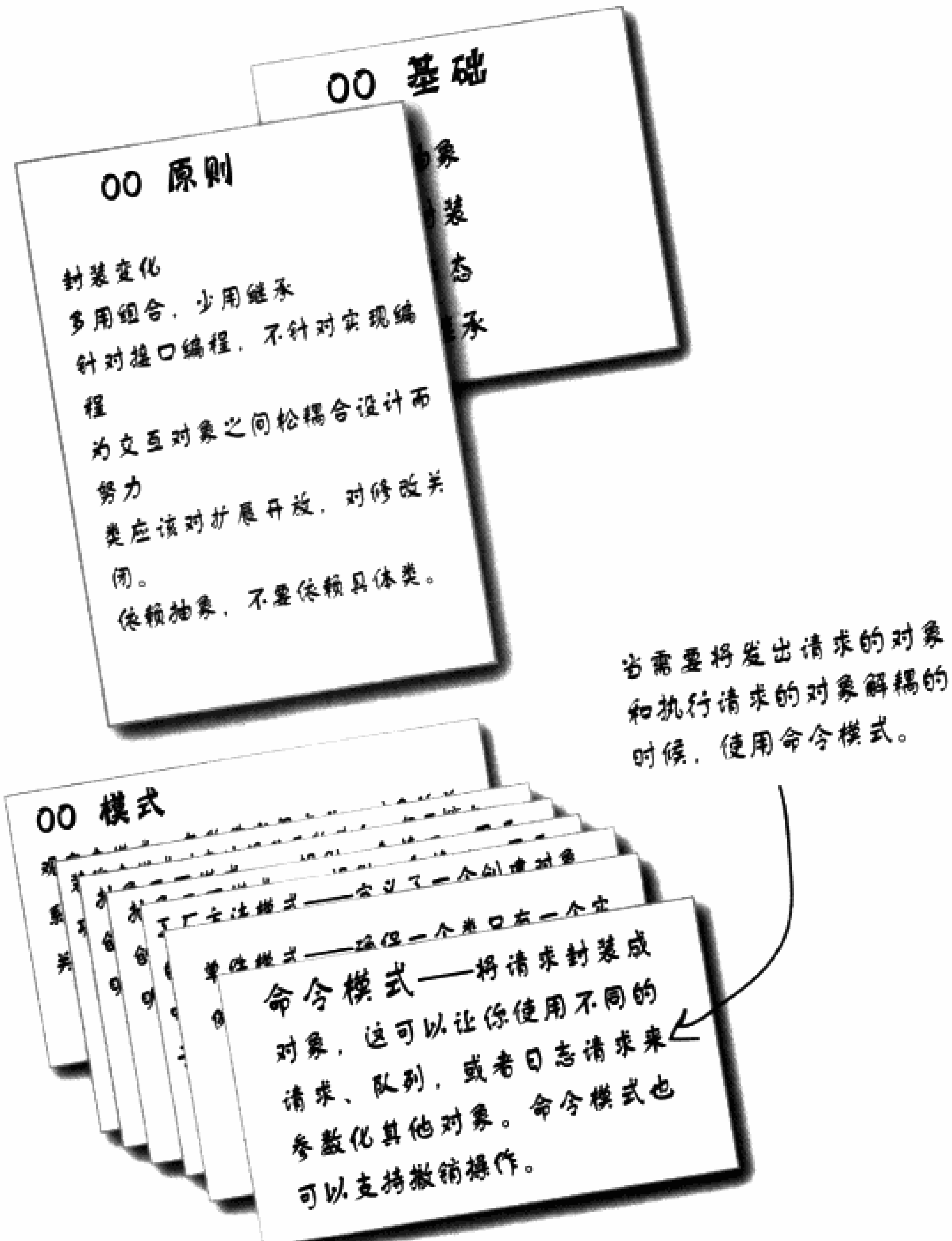
要怎么做呢？当我们执行命令的时候，将历史记录储存在磁盘中。一旦系统死机，我们就可以将命令对象重新加载，并成批地依次调用这些对象的execute()方法。

这种日志的方式对于遥控器来说没有意义，然而，有许多调用大型数据结构的动作的应用无法在每次改变发生时被快速地存储。通过使用记录日志，我们可以将上次检查点（checkpoint）之后的所有操作记录下来，如果系统出状况，从检查点开始应用这些操作。比方说，对于电子表格应用，我们可能想要实现的错误恢复方式是将电子表格的操作记录在日志中，而不是每次电子表格一有变化就记录整个电子表格。对更高级的应用而言，这些技巧可以被扩展应用到事务（transaction）处理中，也就是说，一整群操作必须全部进行完成，或者没有进行任何的操作。



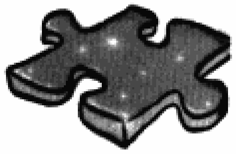
设计箱内的工具

你的工具箱开始变重了！在本章，我们加入了一个模式，这个模式允许我们将动作封装成命令对象，这样一来就可以随心所欲地储存、传递和调用它们。



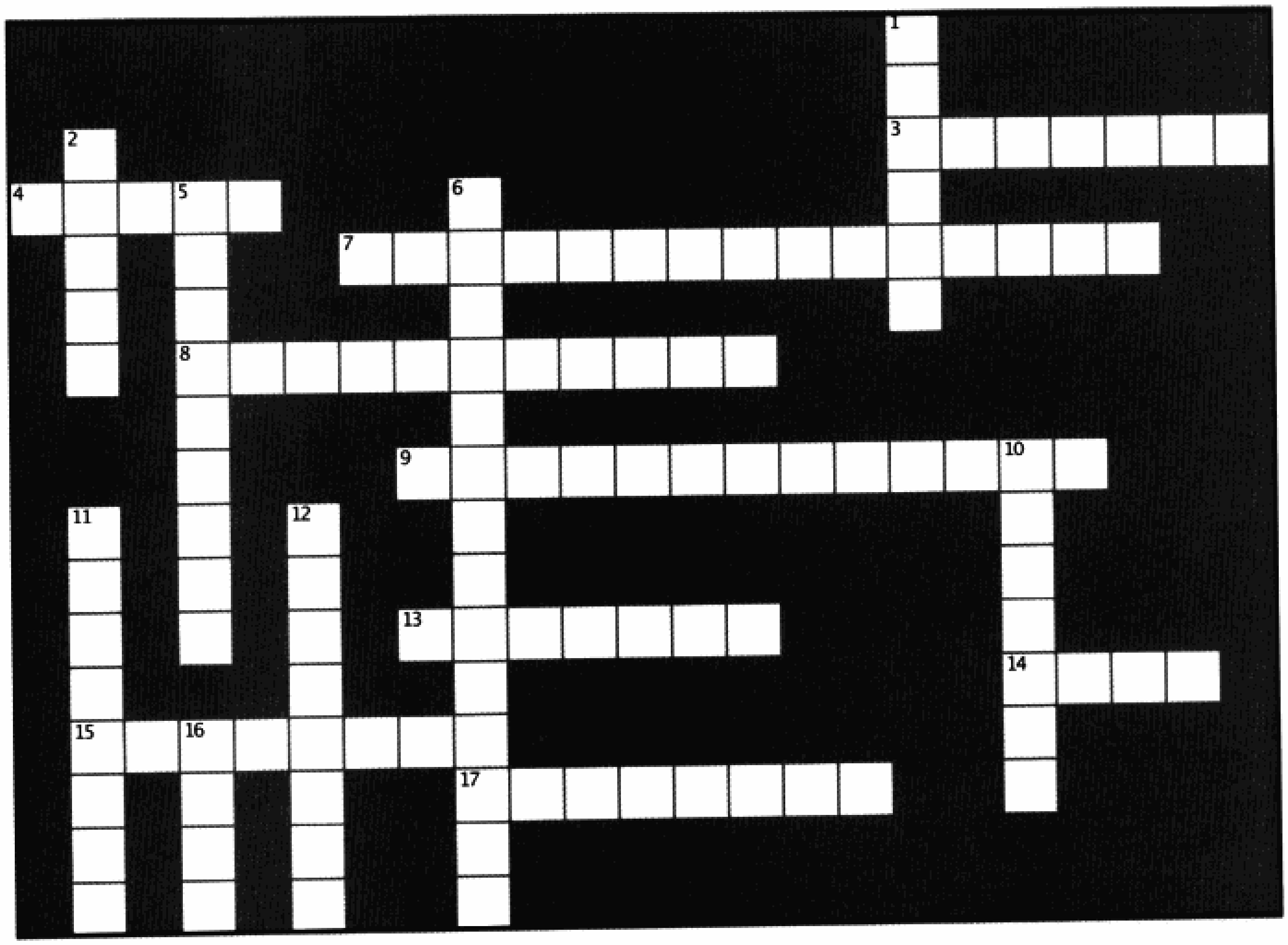
要点

- 命令模式将发出请求的对象和执行请求的对象解耦。
- 在被解耦的两者之间是通过命令对象进行沟通的。命令对象封装了接收者和一个或一组动作。
- 调用者通过调用命令对象的execute()发出请求，这会使得接收者的动作被调用。
- 调用者可以接受命令当做参数，甚至在运行时动态地进行。
- 命令可以支持撤销，做法是实现一个undo()方法来回到execute()被执行前的状态。
- 宏命令是命令的一种简单的延伸，允许调用多个命令。宏方法也可以支持撤销。
- 实际操作时，很常见使用“聪明”命令对象，也就是直接实现了请求，而不是将工作委托给接收者。
- 命令也可以用来实现日志和事务系统。



是时候休息一下了。

这是另一个填字游戏，答案都是来自本章的英文词汇。



横排提示：

3. The Waitress was one
4. A command ____ a set of actions and a receiver
7. Dr. Seuss diner food
8. Our favorite city
9. Act as the receivers in the remote control
13. Object that knows the actions and the receiver
14. Another thing Command can do
15. Object that knows how to get things done
17. A command encapsulates this

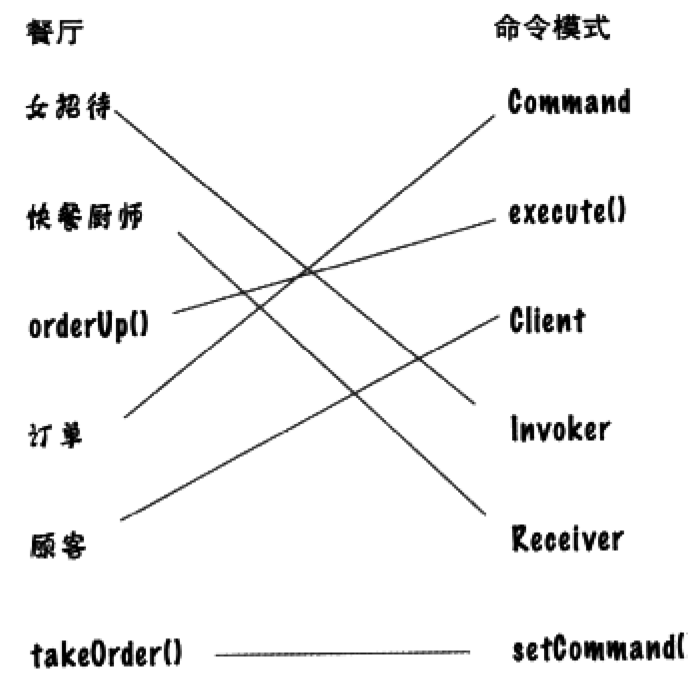
竖排提示：

1. Role of customer in the command pattern
2. Our first command object controlled this
5. Invoker and receiver are ______________
6. Company that got us word of mouth business
10. All commands provide this
11. The cook and this person were definitely decoupled
12. Carries out a request
16. Waitress didn't do this

习题解答

连连看

请将餐厅的对象和方法对应到命令模式的相应名称。



Sharpen your pencil

```
public class GarageDoorOpenCommand implements Command {
    GarageDoor garageDoor;
    public GarageDoorOpenCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }

    public void execute() {
        garageDoor.up();
    }
}
```

```
File Edit Window Help GreenEggs&Ham
%java RemoteControlTest

Light is on
Garage Door is Open

%
```

习题解答

练习

为MacroCommand写下undo()方法。

```
public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

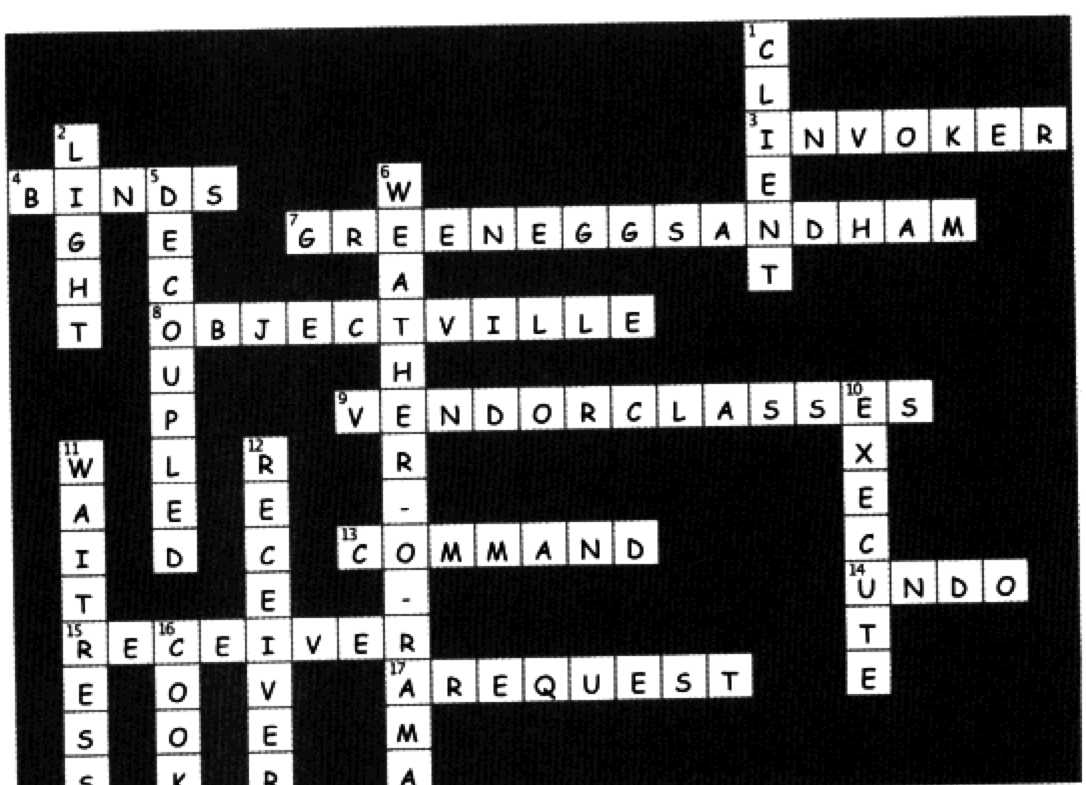
    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].undo();
        }
    }
}
```

Sharpen your pencil

我们也会需要关闭按钮的命令，请在这里写下创建它们的代码：

```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = new HottubOffCommand(hottub);
```



7 适配器模式与外观模式

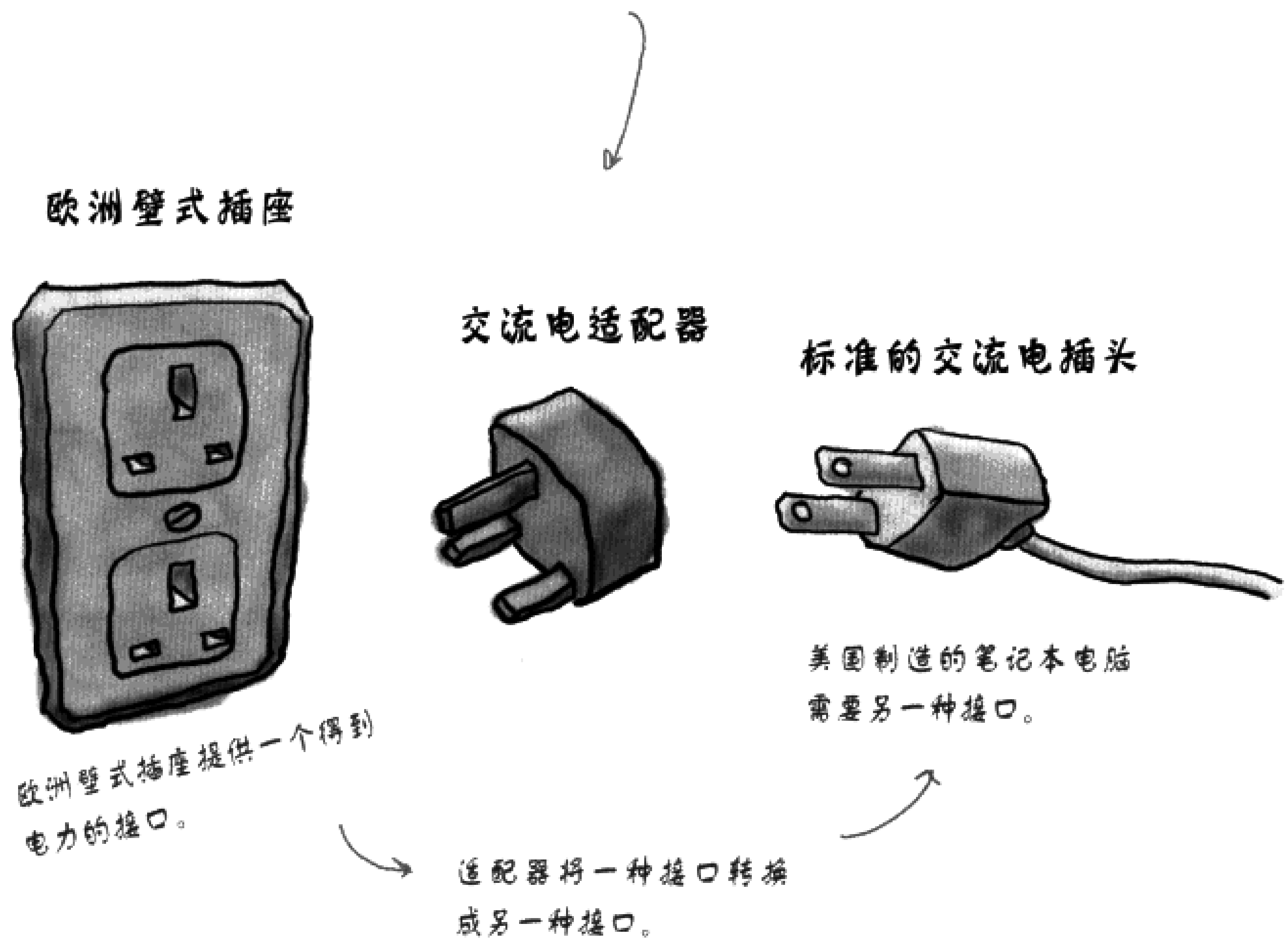
随遇而安



在本章，我们将要进行一项任务，其不可能的程度，简直就像是将一个方块放进一个圆洞中。听起来不可能？有了设计模式，就有可能。还记得装饰者模式吗？我们将对象包装起来，赋予它们新的职责。而现在则是以不同目的，包装某些对象:让它们的接口看起来不像自己而像是别的东西。为何要这样做？因为这样就可以在设计中，将类的接口转换成想要的接口，以便实现不同的接口。不仅如此，我们还要探讨另一个模式，将对象包装起来以简化其接口。

我们周围的适配器

OO适配器是什么，你一定不难理解，因为现实中到处都是。比方说：如果你需要在欧洲国家使用美国制造的笔记本电脑，你可能需要使用一个交流电的适配器……



你知道适配器的作用：它位于美式插头和欧式插座的中间，它的工作是将欧式插座转换成美式插座，好让美式插头可以插进这个插座得到电力。或者也可以这么认为：适配器改变了插座的接口，以符合美式笔记本电脑的需求。

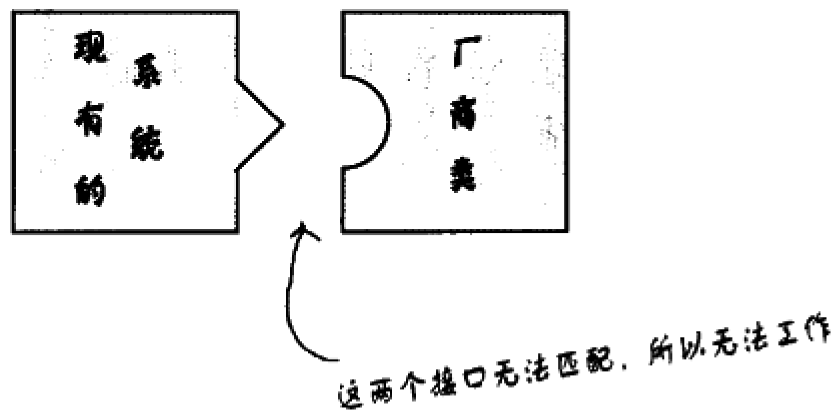
你能想到真实世界中，还有哪些适配器的例子？

某些交流电适配器相当简单，它们只是改变插座的形状来匹配你的插头，直接把电流传送过去。但是有些适配器内部则是相当复杂，可能会改变电流符合装置的需求。

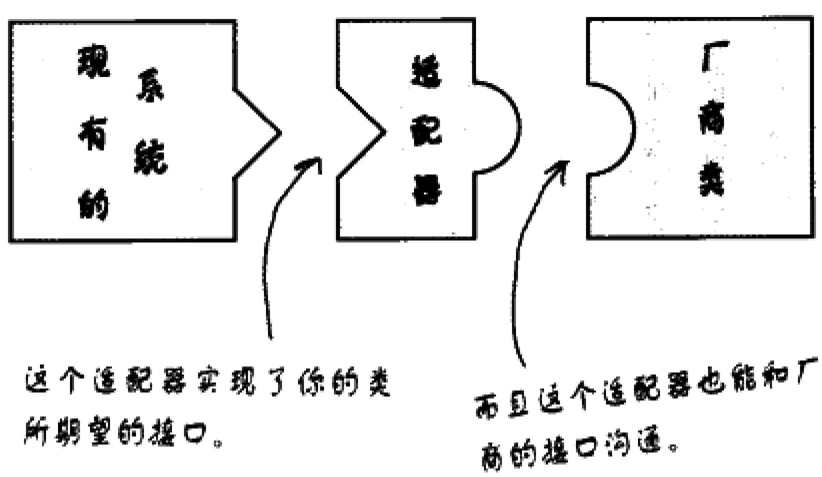
好了，这是真实世界的适配器，那面向对象适配器又是什么？其实，OO适配器和真实世界的适配器扮演着同样的角色：将一个接口转换成另一个接口，以符合客户的期望。

面向对象适配器

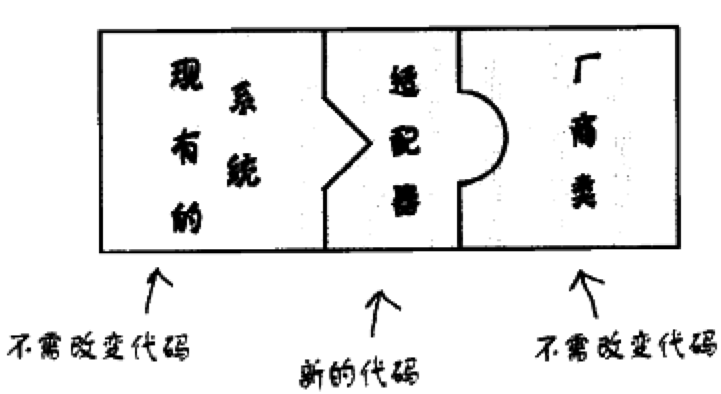
假设已有一个软件系统，你希望它能和一个新的厂商类库搭配使用，但是这个新厂商所设计出来的接口，不同于旧厂商的接口：



你不想改变现有的代码，解决这个问题（而且你也不能改变厂商的代码）。所以该怎么做？这个嘛，你可以写一个类，将新厂商接口转接成你所期望的接口。



这个适配器工作起来就如同一个中间人，它将客户所发出的请求转换成厂商类能理解的请求。



如果它走起路来像只鸭子，叫起来像只鸭子，那么他~~必定~~可能是一只~~鸭子~~包装了鸭子适配器的火鸡……



让我们来看看使用中的适配器。还记得第1章的鸭子吧？让我们看看鸭子接口和类的一个稍微简化的版本：

```
public interface Duck {
    public void quack();
    public void fly();
}
```

这一次，鸭子实现了Duck接口，具备呱呱叫和飞行的能力。

绿头鸭是鸭子的子类。

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly()  {
        System.out.println("I'm flying");
    }
}
```

很简单的实现：只是打印出鸭子在做些什么。

为您介绍最新的“街头顽禽”：

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

火鸡不会呱呱叫，只会咯咯（gobble）叫。

火鸡会飞，虽然飞不远。

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

这是火鸡的一个具体实现。就和鸭子一样，只是打印出火鸡的动作说明。

现在，假设你缺鸭子对象，想用一些火鸡对象来冒充。
显而易见，因为火鸡的接口不同，所以我们不能公然拿来用。

那么，就写个适配器吧：

再靠近一点

首先，你需要实现想转换成的类型接口，也就是你的客户所期望看到的接口。

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

接着，需要取得要适配的对象引用，这里我们利用构造器取得这个引用。

现在我们需要实现接口中所有的方法。quack()在类之间的转换很简单，只要调用gobble()就可以了。

固然两个接口都具备了fly()方法，火鸡的飞行距离很短，不像鸭子可以长途飞行。要让鸭子的飞行和火鸡的飞行能够对应，必须连续五次调用火鸡的fly()来完成。

测试适配器

现在只需要一些代码来测试我们的适配器：

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

先创建一只鸭子……

和一只火鸡。

然后将火鸡包装进一个火鸡适配器中，使它看起来像是一只鸭子。

接着测试这只火鸡：让它咯咯叫，让它飞行。

接着，调用testDuck()方法来测试鸭子，这个方法需要传入一个鸭子对象。

重要的测试来了：我们试着传入一个假装是鸭子的火鸡。

这是我们的testDuck()方法：取得一只鸭子，并调用它的quack()和fly()方法。

测试结果

```
File Edit Window Help Don'tForgetToDuck
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

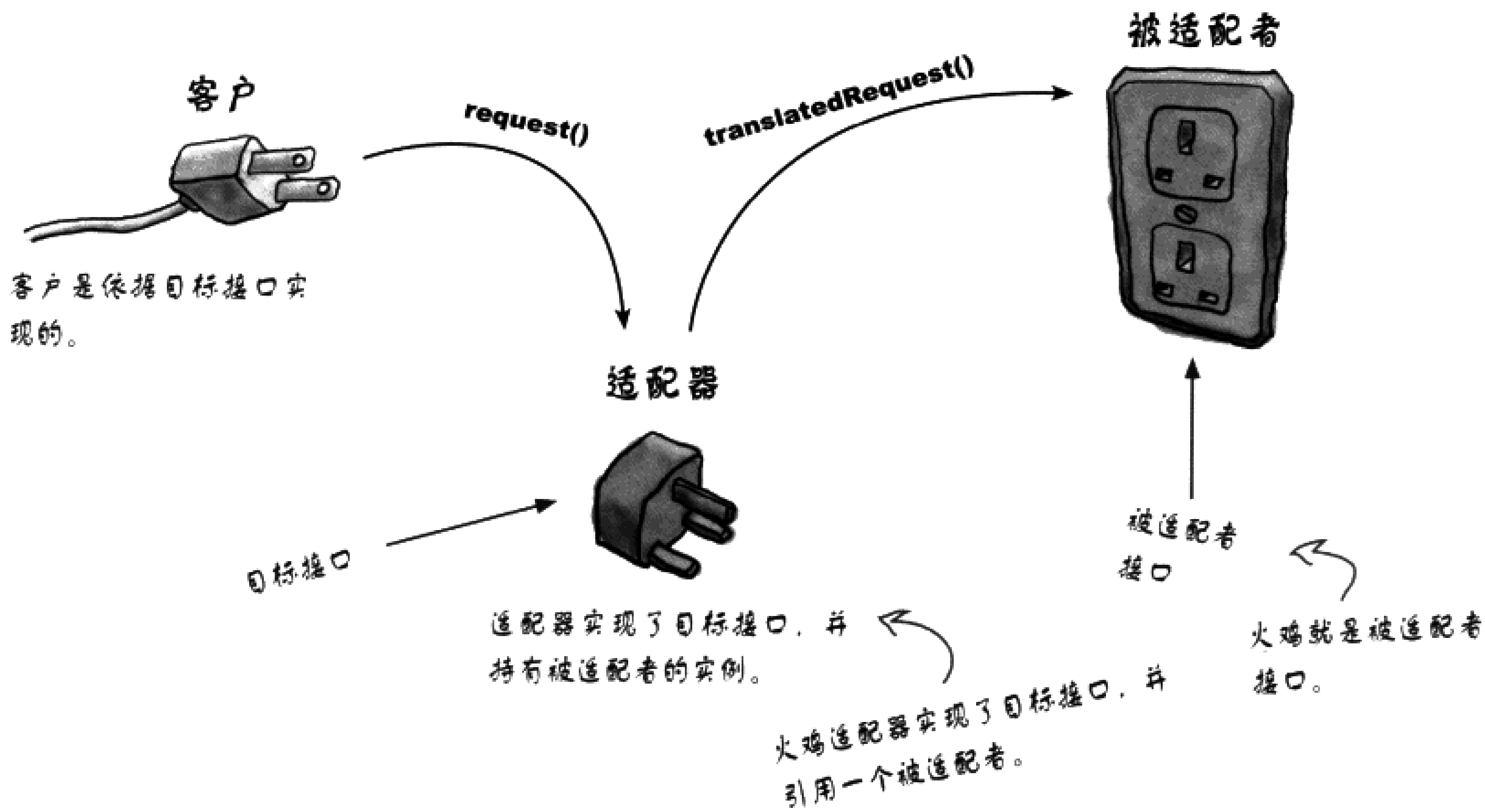
火鸡咯咯叫，且飞行距离短。

鸭子呱呱叫，并能像你期望的那样飞行。

quack()被调用时，适配器咯咯叫；fly()被调用时，适配器飞行了五次。testDuck()方法根本不知道，这其实是一只假装成鸭子的火鸡！

适配器模式解析

现在我们已经知道什么是适配器了，让我们后退一步，再次看看各部分之间的关系。



客户使用适配器的过程如下：

1. 客户通过目标接口调用适配器的方法对适配器发出请求。

2. 适配器使用被适配者接口把请求转换成被适配者的一个或多个调用接口。

3. 客户接收到调用的结果，但并未察觉这一切是适配器在起转换作用。

请注意：客户和被适配者是解耦的，一个不知道另一个。

Sharpen your pencil

如果我们也需要一个将鸭子转换成火鸡的适配器，我们称它为DuckAdapter。请写下这个类：

你如何处理飞行方法（毕竟我们知道鸭子飞得比火鸡远）？答案在本章最后。你认为有更好的方法吗？

there are no Dumb Questions

问： 一个适配器需要做多少“适配”的工作？如果我需要实现一个很大的目标接口，似乎有“很多”工作要做。

答： 的确是如此。实现一个适配器所需要进行的工作，的确和目标接口的大小成正比。如果不用适配器，你就必须改写客户端的代码来调用这个新的接口，将会花许多力气来做大量的调查工作和代码改写工作。相比之下，提供一个适配器类，将所有的改变封装在一个类中，是比较好的做法。

问： 一个适配器只能够封装一个类吗？

答： 适配器模式的工作是将一个接口转换成另一个。虽然大多数的适配器模式所采取的例子都是让一个适配器包装一个被适配者，但我们都知道这个世界其实复杂多了，所以你可能遇到一些状况，需要让一个适配器包装多个被适配者。

这涉及另一个模式，被称为外观模式（Facade Pattern），人们常常将外观模式和适配器模式混为一谈，本章稍后将对此详细说明。

问： 万一我的系统中新旧并存，旧的部分期望旧的厂商接口，但我们却已经使用新厂商的接口编写了这一部分，这个时候该怎么办？这里使用适配器，那里却使用未包装的接口，这实在是让人感到混乱。如果我只是固守着旧的代码，完全不要管适配器，这样子会不会好一些？

答： 不需要如此。可以创建一个双向的适配器，支持两边的接口。想创建一个双向的适配器，就必须实现所涉及的两个接口，这样，这个适配器可以当做旧的接口，或者当做新的接口使用。

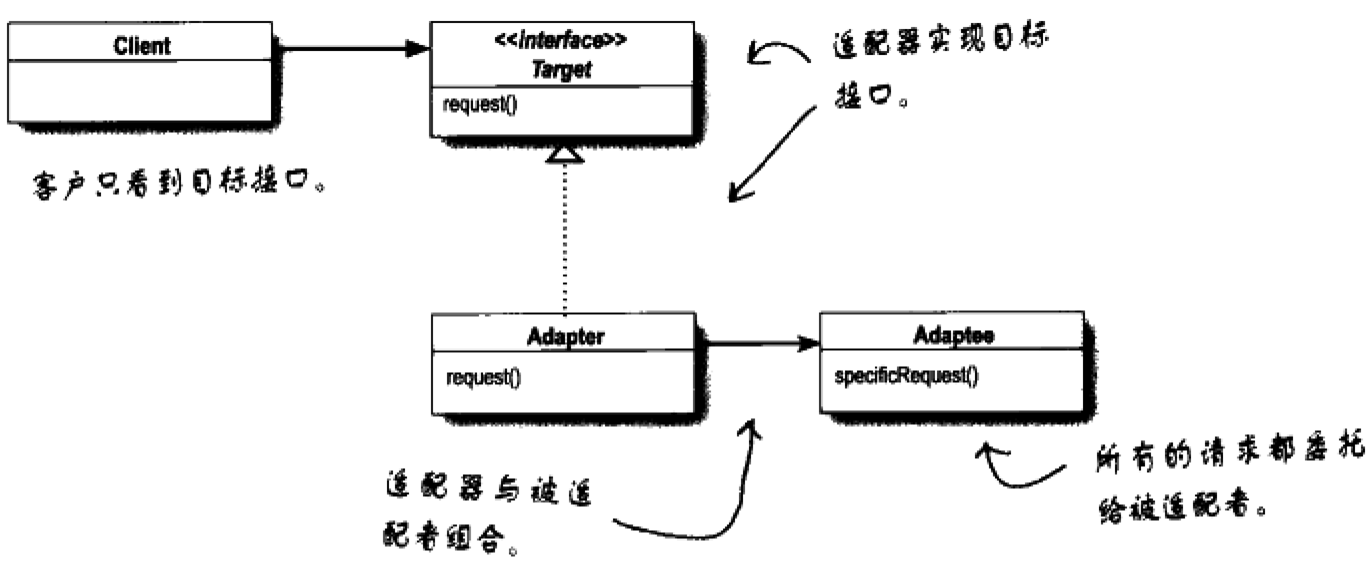
定义适配器模式

玩够了鸭子、火鸡和交流电适配器，现在让我们进入真实世界，并看看适配器模式的正式定义：

> **适配器模式**将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间。

现在，我们知道，这个模式可以通过创建适配器进行接口转换，让不兼容的接口变成兼容。这可以让客户从实现的接口解耦。如果在一段时间之后，我们想要改变接口，适配器可以将改变的部分封装起来，客户就不必为了应对不同的接口而每次跟着修改。

我们已经看过了这个模式的运行时行为，现在来看它的类图：



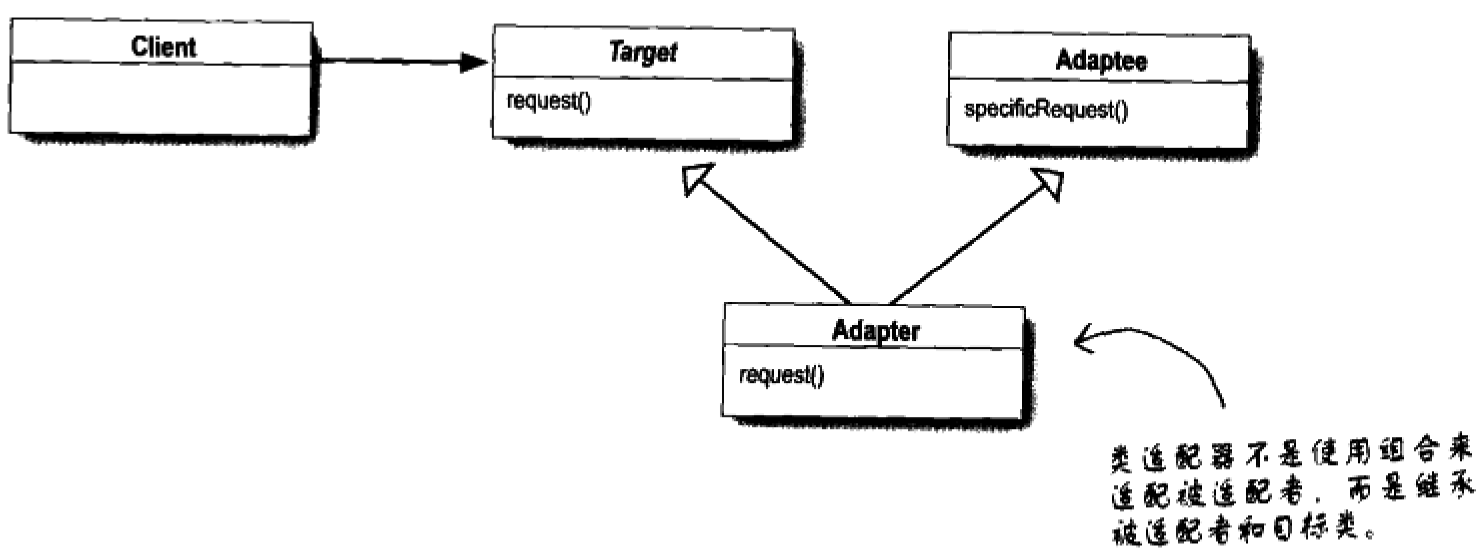
这个适配器模式充满着良好的OO设计原则：使用对象组合，以修改的接口包装被适配者：这种做法还有额外的优点，那就是，被适配者的任何子类，都可以搭配着适配器使用。

也请留意，这个模式是如何把客户和接口绑定起来，而不是和实现绑定起来的。我们可以使用数个适配器，每一个都负责转换不同组的后台类。或者，也可以加上新的实现，只要它们遵守目标接口就可以。

对象和类的适配器

现在，尽管已经定义了这个模式，但其实我们还没有告诉你有关的一切。实际上有“两种”适配器：“对象”适配器和“类”适配器。本章涵盖了对象适配器和类适配器。前一页是对象适配器的图。

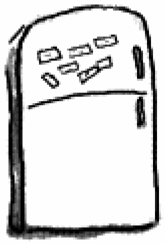
究竟什么是“类”适配器？为什么我们还没告诉你这种适配器？因为你需要多重继承才能够实现它，这在Java中是不可能的。但是当你在使用多重继承语言的时候，还是可能遇到这样的需求。让我们看看多重继承的类图。



看起来很熟悉吗？没错，唯一的差别就在于适配器继承了Target和Adaptee。而对象适配器利用组合的方式将请求传送给被适配者。

BRAIN POWER

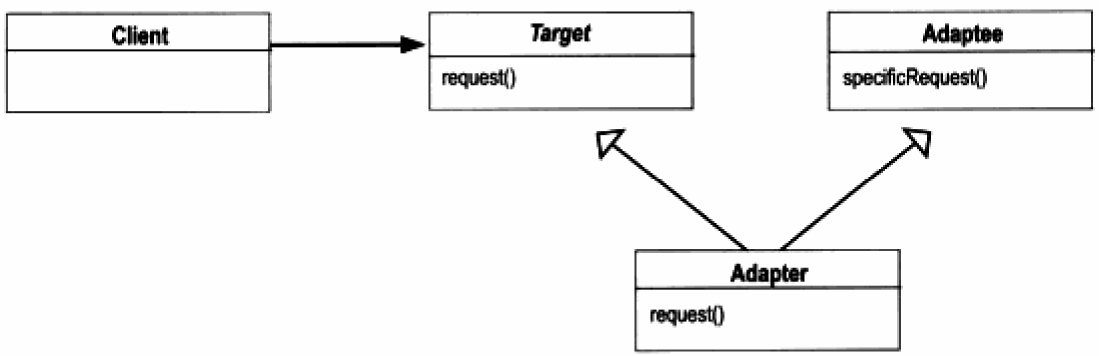
对象适配器和类适配器使用两种不同的适配方法（分别是组合与继承）。这两种实现的差异如何影响适配器的弹性？



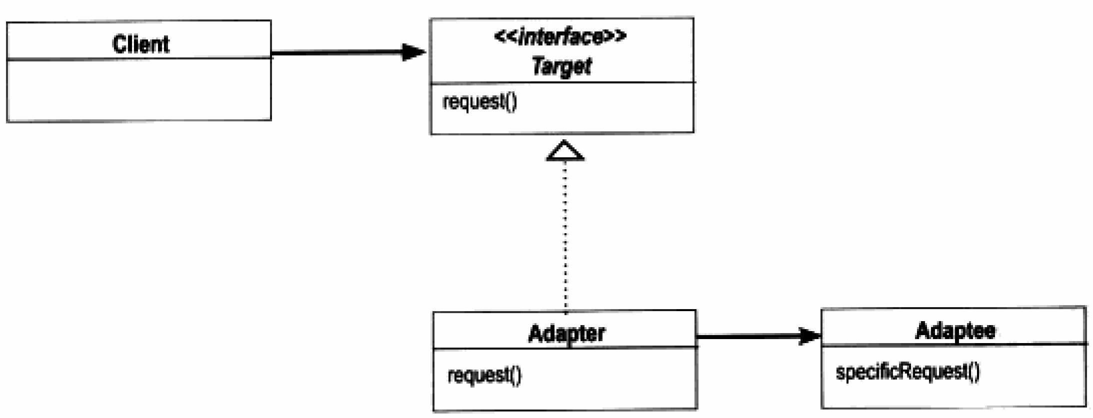
“鸭子”帖

你的任务是把鸭子和火鸡的帖，放置到下图中它们在前面例子里所扮演的角色上。（试着不要翻页看）。然后加上你自己的批注来描述如何工作。

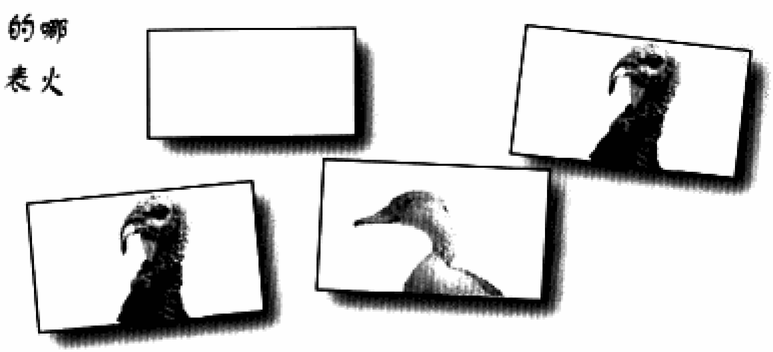
类适配器



对象适配器



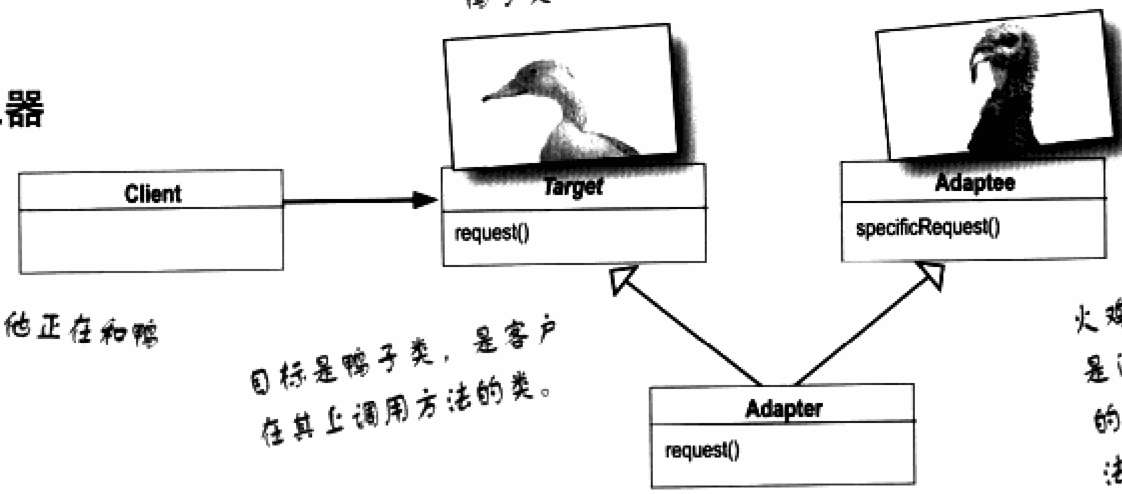
把这些拖到类图上，表示图中的哪一部分代表鸭子，哪一部分代表火鸡。



“鸭子”帖解答

注意一：类适配器使用多重继承，所以你不能在Java中这样做。

类适配器



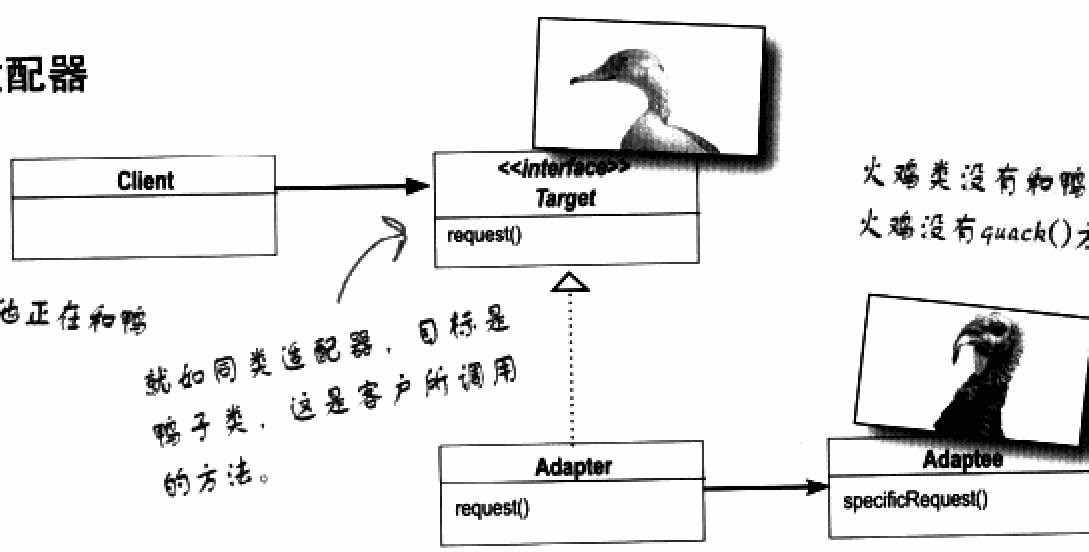
客户认为他正在和鸭子沟通。

目标是鸭子类，是客户在其上调用方法的类。

火鸡类没有鸭子的方法，但是适配器可以将对鸭子方法的调用，转换到调用火鸡方法。

通过扩展两个类（鸭子和火鸡），适配器使得火鸡可以响应对鸭子的请求。

对象适配器



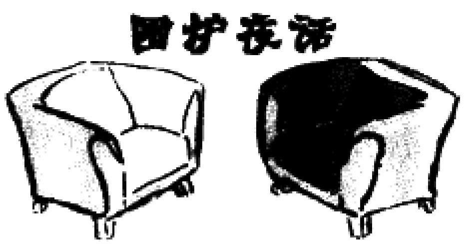
火鸡类没有和鸭子一样的接口，换句话说，火鸡没有quack()方法……

客户认为他正在和鸭子沟通。

就如同类适配器，目标是鸭子类，这是客户所调用的方法。

适配器实现了鸭子的接口，但它收到方法调用时，会委托给火鸡。

感谢适配器，火鸡（被适配者）会接到客户作用于鸭子接口上的调用。



今夜话题：对象适配器和类适配器的面对面接触。

对象适配器

类适配器

对象适配器： 因为我使用组合，我不仅可以适配某个类，也可以适配该类的任何子类，所以我更胜一筹。

类适配器： 你说的是实话，我的确做不到这一点，因为我只能够采用某个特定的被适配类。但是我有一个很大的优点，那就是：我不需要重新实现我的整个被适配者。必要的时候，我也可以覆盖被适配者的行为，因为我利用继承的方式。

对象适配器： 在我的世界中，我们喜欢使用组合多过于使用继承；或许你的做法可以多节省几行代码，但是我只需要写一些代码，将工作委托给被适配者进行。我们喜欢让事情更有弹性。

类适配器： 弹性，或许吧！但效率呢？我可不认为有效率。使用类适配器，仅仅需要一个类适配器，而不需要一个适配器和一个被适配者。

对象适配器： 只不过多了一个小对象，何须如此担心？你或许能够很快地覆盖一个方法，但是我加进适配器代码中的任何行为，都可以和我的被适配者类“以及”其所有的子类搭配工作。

类适配器： 是的，但是万一被适配者的子类加入了新的行为，又会如何？

对象适配器： 嘿！拜托，饶了我吧，我只需要让组合的对象是子类，就可以解决这个问题了。

类适配器： 听起来很麻烦……

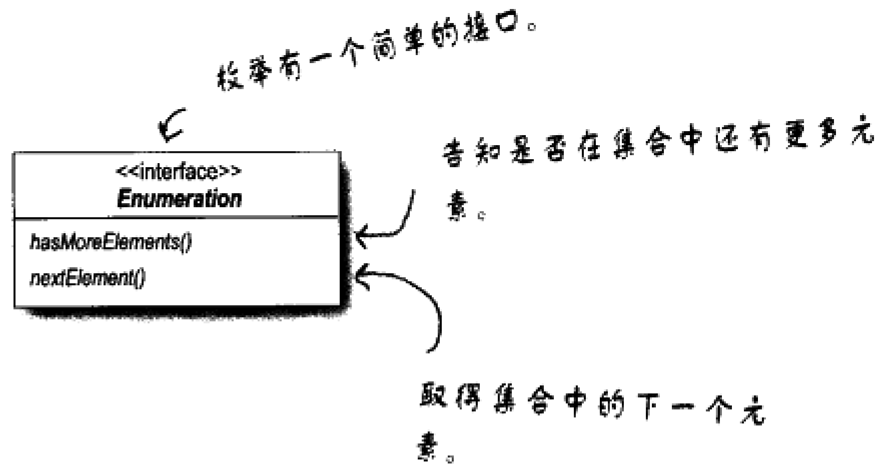
对象适配器： 你想知道麻烦是什么吗？去照照镜子吧！

真实世界的适配器

让我们看看真实世界中一个简单的适配器（至少比鸭子更实际些）……

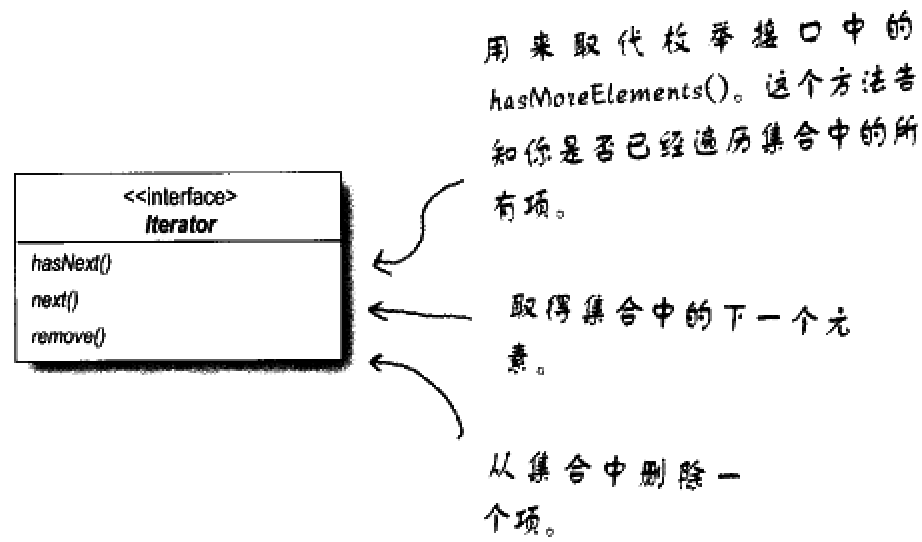
旧世界的枚举器

如果你已经使用过Java，可能记得早期的集合（collection）类型（例如：Vector、Stack、Hashtable）都实现了一个名为elements()的方法。该方法会返回一个Enumeration（举）。这个Enumeration接口可以逐一走过此集合内的每个元素，而无需知道它们在集合内是如何被管理的。



新世界的迭代器

当Sun推出更新后的集合类时，开始使用了Iterator（迭代器）接口，这个接口和枚举接口很像，都可以让你遍历此集合类型内的每个元素，但不同的是，迭代器还提供了删除元素的能力。

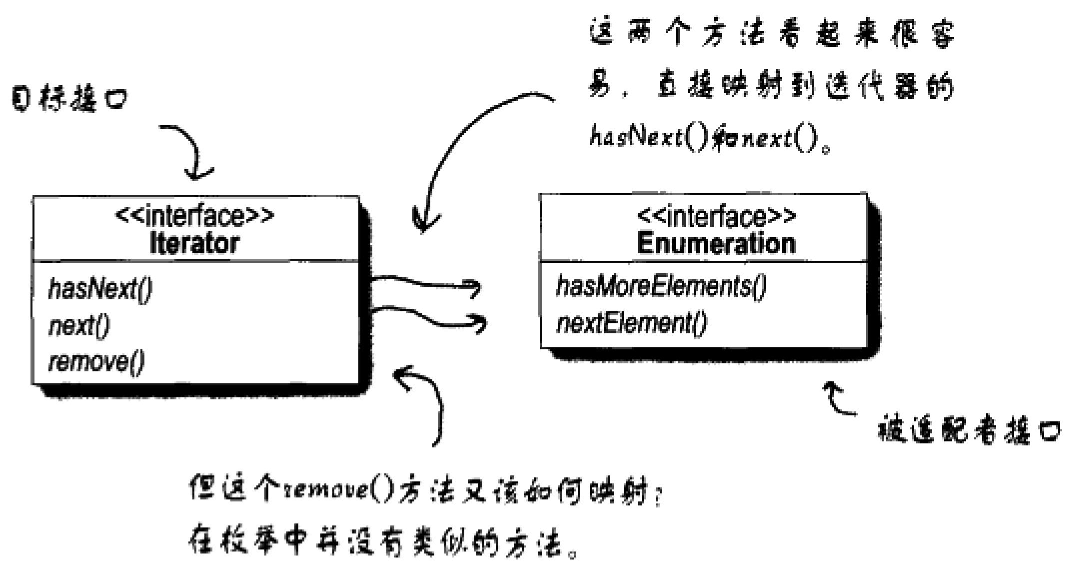


而今天……

我们经常面对遗留代码，这些代码暴露出枚举器接口，但我们又希望在新的代码中只使用迭代器。想解决这个问题，看来我们需要构造一个适配器。

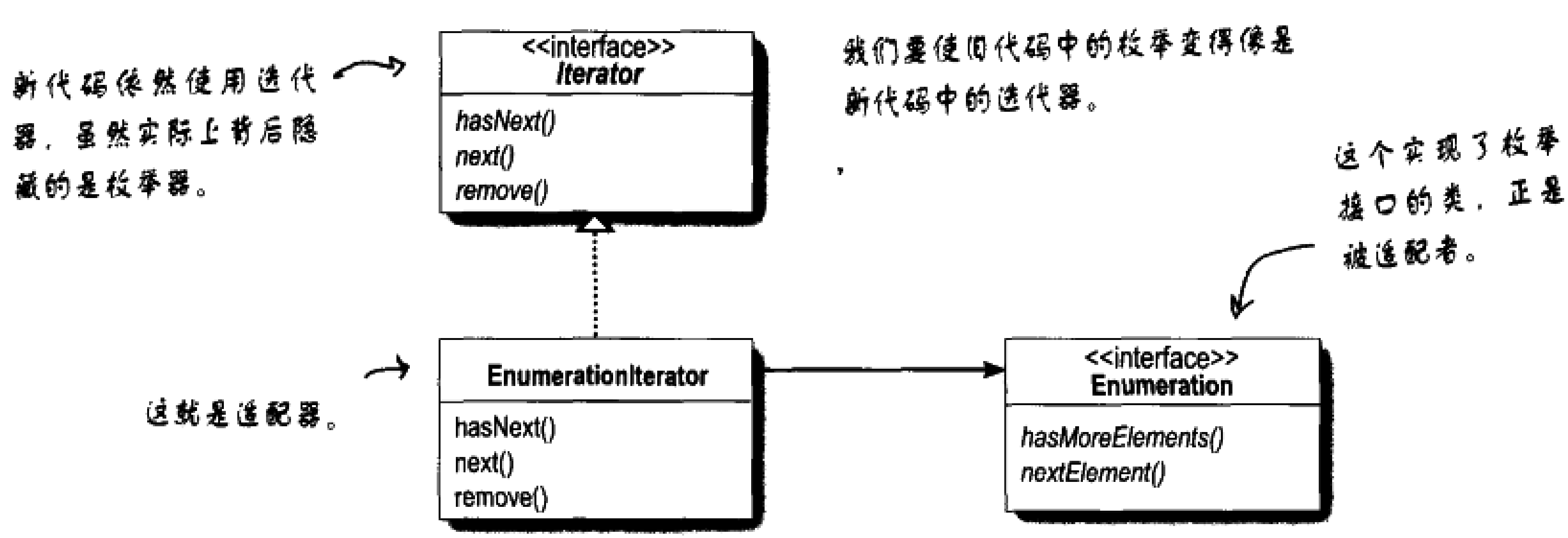
将枚举适配到迭代器

我们先看看这两个接口，找出它们的方法映射关系。换句话说，我们要找出每一个适配器方法在被适配者中的对应方法是什么。



设计适配器

这个类应该是这样的：我们需要一个适配器，实现了目标接口，而此目标接口是由被适配者所组合的。hasNext()和next()方法很容易实现，直接把它们从目标对应到被适配者就可以了。但是对于remove()方法，我们又该怎么办？请花一些时间想一想（我们在下一页就会处理）。目前，类图是这样的：



处理remove()方法

好了，我们知道枚举不支持删除，因为枚举是一个"只读"接口。适配器无法实现一个有实际功能的remove()方法，最多只能抛出一个运行时异常。幸运地，迭代器接口的设计者事先料到了这样的需要，所以将remove()方法定义成会抛出UnsupportedOpeartionException。

在这个例子中，我们看到了适配器并不完美；客户必须小心潜在的异常，但只要客户够小心，而且适配器的文档能做出说明，这也算是一个合理的解决方案。

编写一个EnumeratorIterator适配器

这是一份简单而有效的代码，适合依然会产生枚举的遗留类。

```
public class EnumerationIterator implements Iterator
{
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }

    public boolean hasNext() {
        return enum.hasMoreElements();
    }

    public Object next() {
        return enum.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

因为我们将枚举适配成迭代器，适配器需要实现迭代器接口……适配器必须看起来就像是一个迭代器。

我们利用组合的方式，将枚举结合进入适配器中，所以用一个实例变量记录枚举。

迭代器的hasNext()方法其实是委托给枚举的hasMoreElements()方法……

……而迭代器的next()方法其实是委托给枚举的nextElement()方法。

很不幸，我们不能支持迭代器的remove()方法，所以必须放弃。在这里，我们的做法是抛出一个异常。

练习

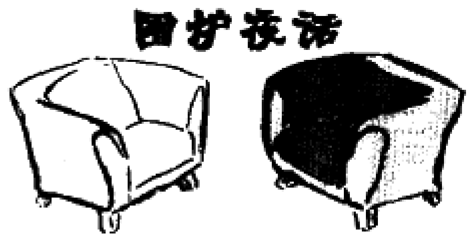
虽然Java已经采用了迭代器，但还是有相当多的遗留“客户代码”，依赖于枚举接口，所以利用适配器将迭代器转换成枚举，其实是很有用的技巧。

编写一个适配器来做这样的转换，可以将此适配器用在ArrayList上作为测试。ArrayList类支持迭代器接口，但不支持枚举（尚未支持）。

BRAIN POWER

某些交流电适配器所做的事情不只是改变接口，它们还加了一些其他的特性，例如：电涌保护、指示灯、警报声等。

如果要你实现这类特性，你要使用什么模式？



今夜话题：装饰者模式和适配器模式讨论彼此的差异。

装饰者	适配器
我很重要，我的工作全都是和“责任”相关的。你知道的，当事情一涉及到装饰者，就表示有一些新的行为或责任要加入到你的设计中。	
	你们这些家伙老是把光环放在自己身上，但我们这些适配器却隐身于沟渠中，干着脏活——转换接口。我们的工作或许不是光彩夺目，但我们的客户却很感激我们让他们的生活变得更容易。
你说的可能是真的，但可不要认为我们工作不努力。当我们必须装饰一个大型接口时，咳！可是需要很多代码的。	
	当你必须将若干类整合在一起来提供你的客户所期望的接口时，不妨扮演适配器的角色看看，这才够棘手。不过，我们有一句格言：“被解耦的客户才是快乐的客户”。
很俏皮！别认为我们独揽了所有的光环，有时候我只是一个装饰者，天晓得还有多少其他的装饰者会再将我包装起来。当一个方法调用委托给我时，我根本不知道有多少其他装饰者已经处理过这个调用了，而我也根本不知道我对这个请求所做的付出是否会得到别人的注意。	
	哎呀，我们其实同病相怜。只要适配器工作顺利，客户甚至不会意识到我们的存在。根本没有人会感谢适配器所做的一切。

装饰者

适配器

但是，关于我们适配器的好处是，我们允许客户使用新的库和子集合，无须改变“任何”代码，由我们负责做转换即可。嘿！这是我们的市场。

我们装饰者也可以做到，但是我们可以让“新行为”加入类中，而无需修改现有的代码。我还是认为适配器只是一种装饰者的变体，我的意思是说，适配器就和我们一样，都是用来包装对象的。

不！不！不！才不是这样。我们“一定会”进行接口的转换，但你们“绝不会”这么做。我宁可认为装饰者其实是一种适配器的变体；只是你们不会改变接口。

不！我们的工作是扩展我们包装的对象的行为或责任，并不是“简单传送”就算了。

嘿！你说谁“简单传送”？来呀！转换几个接口让我瞧瞧，看你能持续多久！

或许我们应该体会到，我们在纸上看起来虽然很类似，但其实我们的意图差异颇大。

没错，你这么说就对了。

现在，看看不同之处……

在本章中，还有另一个模式。

你已经知道适配器模式是如何将一个类的接口转换成另一个符合客户期望的接口的。你也知道在Java中要做到这一点，必须将一个不兼容接口的对象包装起来，变成兼容的对象。

我们现在要看一个改变接口的新模式，但是它改变接口的原因是为了简化接口。这个模式被巧妙地命名为外观模式（Facade-Pattern），之所以这么称呼，是因为它将一个或数个类的复杂的一切都隐藏在背后，只显露出一个干净美好的外观。

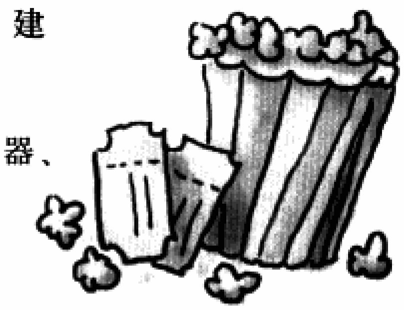
连连看

找出每个模式的目的：

模式	意图
装饰者	将一个接口转成另一个接口
适配器	不改变接口，但加入责任
外观	让接口更简单

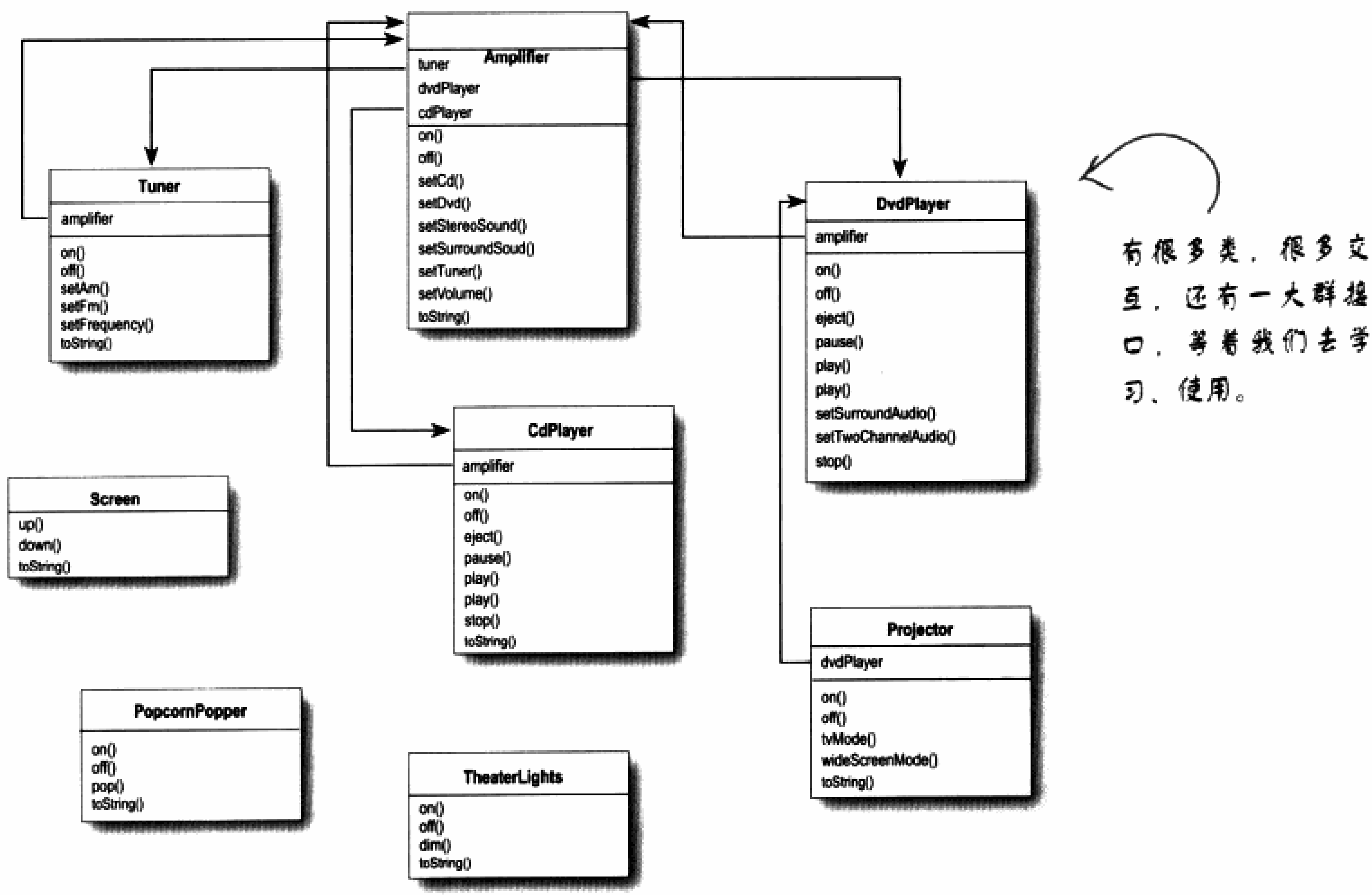
甜蜜的家庭影院

在我们进入外观模式的细节之前，让我们看一个风行全美的热潮：建立自己的家庭影院。



通过一番研究比较，你组装了一套杀手级的系统，内含DVD播放器、投影机、自动屏幕、环绕立体声，甚至还有爆米花机。

看看这些组件的组成：



你花了好几个星期布线、挂上投影机、连接所有的装置并进行微调。现在，你准备开始享受一部电影……

观赏电影（困难的方式）

挑选一部DVD影片，放松，准备开始感受电影的魔幻魅力。
哎呀！忘了一件事：想看电影，必须先执行一些任务。

1. 打开爆米花机
2. 开始爆米花
3. 将灯光调暗
4. 放下屏幕
5. 打开投影机
6. 将投影机的输入切换到DVD
7. 将投影机设置在宽屏模式
8. 打开功放
9. 将功放的输入设置为DVD
10. 将功放设置为环绕立体声
11. 将功放音量调到中（5）
12. 打开DVD播放器
13. 开始播放DVD



让我们将这些任务写成类和方法的调用

```
popper.on();
popper.pop();

lights.dim(10);

screen.down();

projector.on();
projector.setInput(dvd);
projector.wideScreenMode();

amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);

dvd.on();
dvd.play(movie);
```

涉及到六个不同的类！

打开爆米花机，开始爆米花。

灯光调暗到10%的亮度……

把屏幕放下。

打开投影机，并将它设置在宽屏模式……

打开功放，设置为DVD，调整成环绕立体声模式，音量调到5……

打开DVD播放机……“终于”可以看电影了！

但还不只这样……

- 看完电影后，你还要把一切都关掉，怎么办？难道要反向地把这一切动作再进行一次？
- 如果要听CD或者广播，难道也会这么麻烦？
- 如果你决定要升级你的系统，可能还必须重新学习一套稍微不同的操作过程。

怎么办？使用你的家庭影院竟变得如此复杂！让我们看看外观模式如何解决这团混乱，好让你能轻易地享受电影……

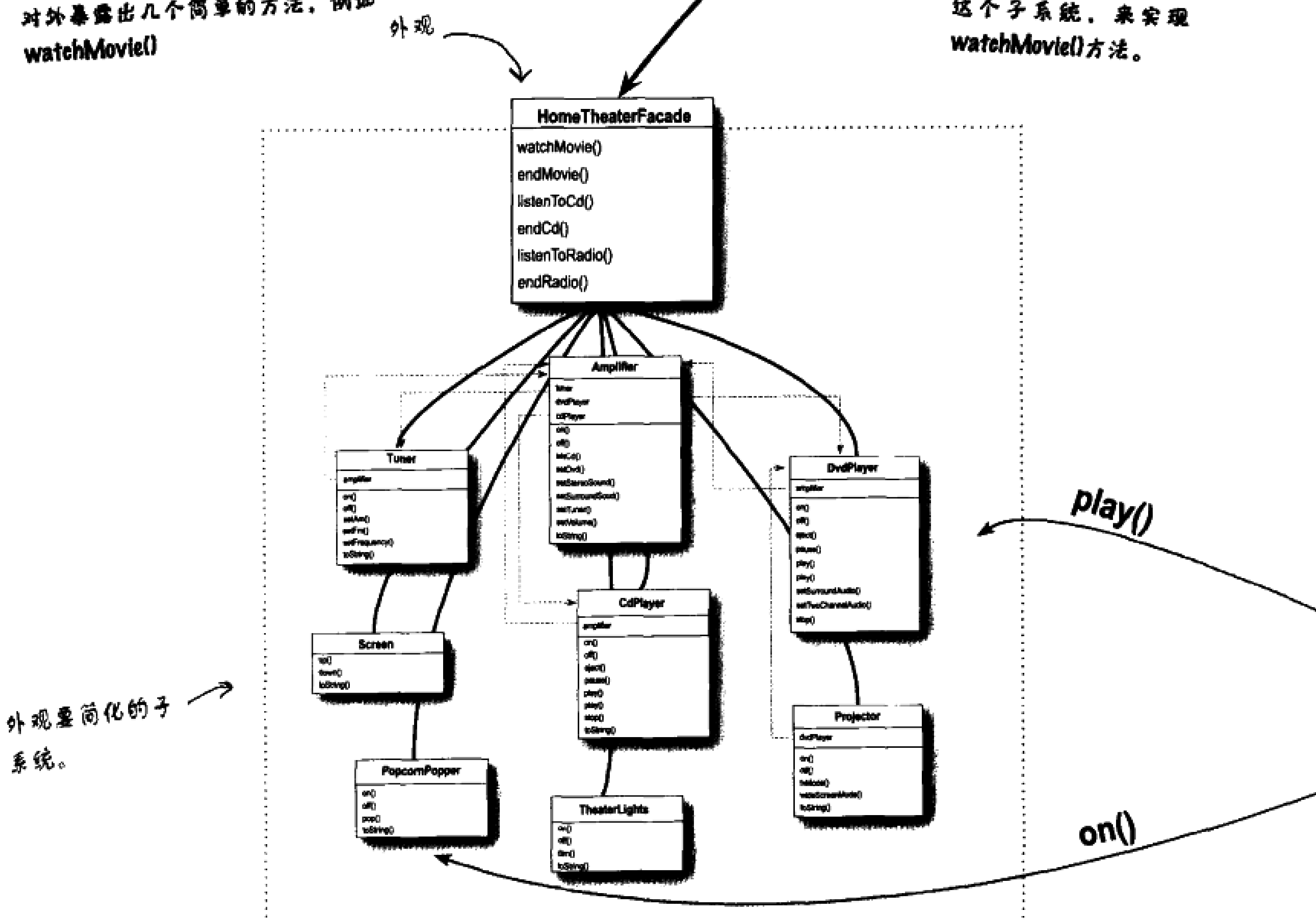
灯光、相机、外观！

你需要的正是一个外观：有了外观模式，通过实现一个提供更合理的接口的外观类，你可以将一个复杂的子系统变得容易使用。如果你需要复杂子系统的强大威力，别担心，还是可以使用原来的复杂接口的；但如果你需要的是一个方便使用的接口，那就使用外观。

让我们看看外观如何运作：

❶ 现在是为家庭影院系统创建一个外观的时候了，于是我们创建了一个名为HomeTheaterFacade的新类，它对外暴露出几个简单的方法，例如watchMovie()

❷ 这个外观类将家庭影院的诸多组件视为一个子系统，通过调用这个子系统，来实现watchMovie()方法。



watchMovie()

这是子系统外观的客户端。



3 现在，你的客户代码可以调用此家庭影院外观所提供的方法，而不必再调用这个子系统的方法。所以，想要看电影，我们只要调用一个方法（也就是watchMovie()）就可以了。灯光、DVD播放器、投影机、功放、屏幕、爆米花，一口气全部搞定。

我就是喜欢接触这些低层的操作！



这是旧鸭中学的影音科学社前任社长。

4 外观只是提供你更直接的操作，并未将原来的子系统阻隔起来。如果你需要子系统类的更高层功能，还是可以使用原来的子系统。

there are no Dumb Questions

问： 如果外观封装了子系统的类，那么需要低层功能的客户如何接触这些类？

答： 外观没有"封装"子系统的类，外观只提供简化的接口。所以客户如果觉得有必要，依然可以直接使用子系统的类。这是外观模式一个很好的特征：提供简化的接口的同时，依然将系统完整的功能暴露出来，以供需要的人使用。

问： 外观会新增功能吗，或者它只是将每一个请求转由子系统执行？

答： 外观可以附加"聪明的"功能，让使用子系统更方便。比方说，虽然你的家庭影院外观没有实现任何新行为，但是外观却够聪明，知道爆米花机要先开启然后才能开始爆米花（同样，也要先开机才能放电影）。

问： 每个子系统只能有一个外观吗？

答： 不，你可以为一个子系统创建许多个外观。

问： 除了能够提供一个比较简单的接口之外，外观模式还有其他的优点吗？

答： 外观模式也允许你将客户实现从任何子系统中解耦。比方说，你得到了大笔加薪，所以想要升级你的家庭影院，采用全新的和以前不一样接口的组件。如果当初你的客户代码是针对外观而不是针对子系统编写的，现在你就不需要改变客户代码，只需要修改外观代码（而且有可能厂商会提供新版的外观代码）。

问： 我可不可以这样说，适配器模式和外观模式之间的差异在于：适配器包装一个类，而外观可以代表许多类？

答： 不对！提醒你，适配器模式将一个或多个类接口变成客户所期望的一个接口。虽然大多数教科书所采用的例子中适配器只适配一个类，但是你可以适配许多类来提供一个接口让客户编码。类似地，一个外观也可以只针对一个拥有复杂接口的类提供简化的接口。两种模式的差异，不在于它们"包装"了几个类，而是在于它们的意图。适配器模式的意图是，"改变"接口符合客户的期望；而外观模式的意图是，提供子系统的一个简化接口。

外观不只是简化了接口，也将客户从组件的子系统中解耦。

外观和适配器可以包装许多类，但是外观的意图是简化接口，而适配器的意图是将接口转换成不同接口。

构造家庭影院外观

让我们逐步构造家庭影院外观：第一步是使用组合让外观能够访问子系统中所有的组件。

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                 Tuner tuner,
                 DvdPlayer dvd,
                 CdPlayer cd,
                 Projector projector,
                 Screen screen,
                 TheaterLights lights,
                 PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // 其他的方法

}
```

这就是组合：我们会用到的子系统组件全部都在这里。

外观将子系统中每一个组件的引用都传入它的构造器中。然后外观把它们赋值给相应的实例变量。

这部分的代码，等一下就会填进去……

实现简化的接口

现在该是时候将子系统的组件整合成一个统一的接口了。让我们实现watchMovie()和endMovie()两个方法：

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

watchMovie()将我们之前手动进行的每项任务依次处理。请注意，每项任务都是委托子系统中相应的组件处理的。

```
public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

而endMovie()负责关闭一切。每项任务也都是委托子系统中合适的组件处理的。

BRAIN POWER

想想看，你在JavaAPI中遇到过哪些外观，你还希望Java能够新增哪些外观？

观赏电影（用轻松的方式）



这是大显身手的时刻！

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // 在这里实例化组件

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

我们就在这个测试程序中，直接建立了这些组件。正常的情况下，某个外观会被指派给客户使用，而不需要由客户自行创建外观。

首先，根据子系统所有的组件来实例化外观。

使用简化的接口，先开启电影，然后关闭电影。

输出结果是这样的：

只要调用外观的watchMovie()，就一切搞定了……

```
File Edit Window Help SnakesWhy'dItHaveToBeSnakes?
%java HomeTheaterTestDrive
Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

……在这里，我们已经看完电影了，所以调用endMovie()将一切都关闭。

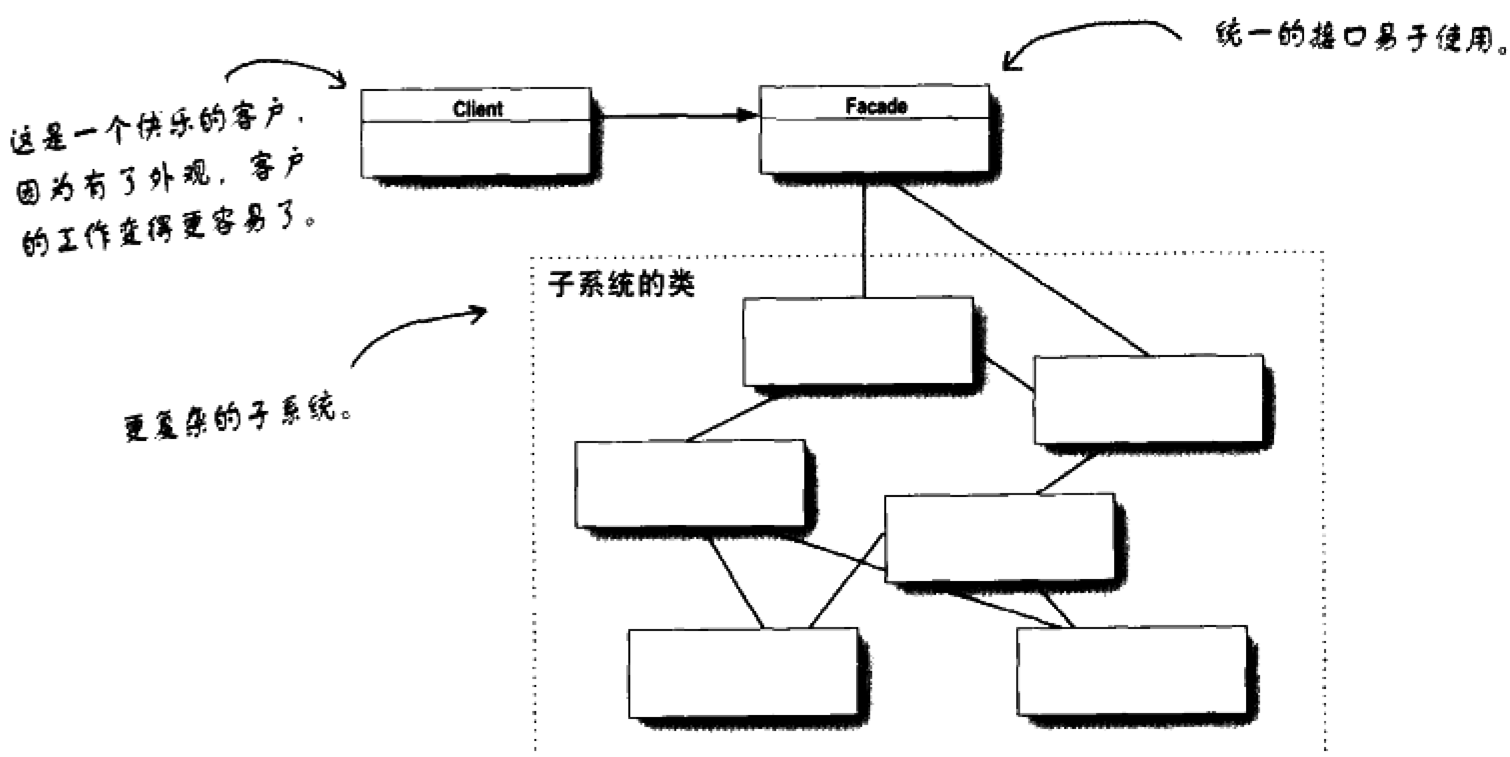
定义外观模式

想要使用外观模式，我们创建了一个接口简化而统一的类，用来包装子系统中一个或多个复杂的类。外观模式相当直接，很容易理解，这方面和许多其他的模式不太一样。但这并不会降低它的威力：外观模式允许我们让客户和子系统之间避免紧耦合，而且稍后你还会看到，外观模式也可以帮我们遵守一个新的面向对象原则。

在介绍这个新的原则之前，先来看看外观模式的正式定义：

> **外观模式**提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

这很容易理解，但是请务必记得模式的意图。这个定义清楚地告诉我们，外观的意图是要提供一个简单的接口，好让一个子系统更易于使用。从这个模式的类图可以感受到这一点：



全部内容就是这样，你又多学会了一个模式！现在来看一个新的OO原则。请注意，这个原则可能有点挑战性！

“最少知识”原则

最少知识（Least Knowledge）原则告诉我们要减少对象之间的交互，只留下几个“密友”。这个原则通常是这么说的：

设计原则

最少知识原则：只和你的密友谈话。

这到底是什么意思？这是说，当你正在设计一个系统，不管是任何对象，你都要注意它所交互的类有哪些，并注意它和这些类是如何交互的。

这个原则希望我们在设计中，不要让太多的类耦合在一起，免得修改系统中一部分，会影响到其他部分。如果许多类之间相互依赖，那么这个系统就会变成一个易碎的系统，它需要花许多成本维护，也会因为太复杂而不容易被其他人了解。

BRAIN POWER

这段代码耦合了多少类？

```
public float getTemp() {
    return station.getThermometer().getTemperature();
}
```

如何不要赢得太多的朋友和影响太多的对象

究竟要怎样才能避免这样呢？这个原则提供了一些方针：就任何对象而言，在该对象的方法内，我们只应该调用属于以下范围的方法：

- 该对象本身
- 被当做方法的参数而传递进来的对象
- 此方法所创建或实例化的任何对象
- 对象的任何组件

请注意，这些方针告诉我们，如果某对象是调用其他的方法的返回结果，不要调用该对象的方法！

把“组件”想象成是被实例变量所引用的任何对象，换句话说，把这想象成是“有一个”（HAS-A）关系。

这听起来有点严厉，不是吗？如果调用从另一个调用中返回的对象的方法，会有什么害处呢？如果我们这样做，相当于向另一个对象的子部分发请求（而增加我们直接认识的对象数目）。在这种情况下，原则要我们改为要求该对象为我们做出请求，这么一来，我们就不需要认识该对象的组件了（让我们的朋友圈子维持在最小的状态）。比方说：

不采用这个原则

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

这里，我们从气象站取得了温度计（thermometer）对象，然后再从温度计对象取得温度。

采用这个原则

```
public float getTemp() {
    return station.getTemperature();
}
```

应用此原则时，我们在气象站中加进一个方法，用来向温度计请求温度。这可以减少我们所依赖的类的数目。

将方法调用保持在界限内……

这是一个汽车类，展示调用方法的各种做法，同时还能够遵守最少知识原则：

```
public class Car {
        Engine engine;
        // 其他实例变量

        public Car() {
                // 初始化发动机
        }

        public void start(Key key) {
                Doors doors = new Doors();

                boolean authorized = key.turns();

                if (authorized) {
                        engine.start();
                        updateDashboardDisplay();
                        doors.lock();
                }
        }

        public void updateDashboardDisplay() {
                // 更新显示
        }
}
```

这是类的一个组件，我们能够调用它的方法。

在这里创建了一个新的对象，它的方法可以被调用。

被当做参数传进来的对象，其方法可以被调用。

可以调用对象组件的方法。

可以调用同一个对象内的本地方法（local method）。

可以调用你所创建或实例化的对象的方法。

there are no Dumb Questions

问：还有另一个原则，叫做得墨忒耳法则（Law of Demeter），它和最少知识原则有什么关系？

答： 其实两个名词指的是同一个原则。我们倾向于使用最少知识原则来称呼它是因为以下两个原因：(1)这个名字更直接。(2)法则（Law）给人的感觉是强制的。事实上，没有任何原则是法律（law），所有的原则都应该在有帮助的时候才遵守。所有的设计都不免需要折衷（在抽象和速度之间取舍，在空间和时间之间平衡……）。虽然原则提供了方针，但在采用原则之前，必须全盘考虑所有的因素。

问：采用最少知识原则有什么缺点吗？

答： 是的，虽然这个原则减少了对象之间的依赖，研究显示这会减少软件的维护成本；但是采用这个原则也会导致更多的“包装”类被制造出来，以处理和其他组件的沟通，这可能会导致复杂度和开发时间的增加，并降低运行时的性能。

Sharpen your pencil

这些类有没有违反最少知识原则？请说明原因。

```
public House {
    WeatherStation station;

    // 其他的方法和构造器

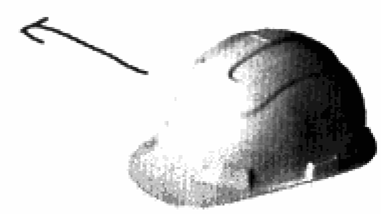
    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

```
public House {
     WeatherStation station;

    // 其他的方法和构造器

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```



施工区！
注意落物伤人

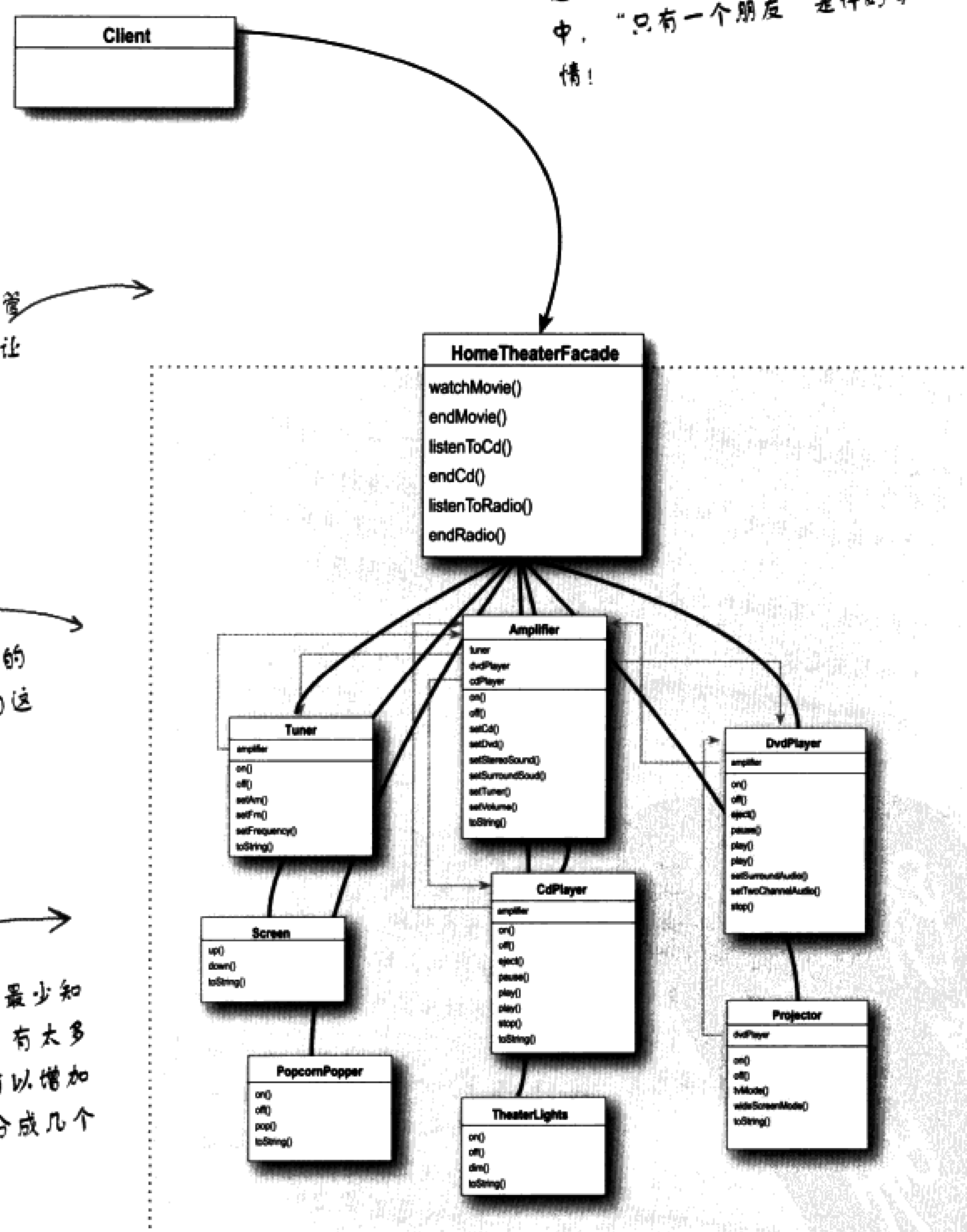
BRAIN POWER

你能够想出在Java中，有哪些常用的地方违反了最少知识原则吗？
你应该注意吗？

回答：你认为System.out.println()如何？

外观和最少知识原则

这个客户只有一个朋友，也就是HomeTheaterFacade。在OO编程中，“只有一个朋友”是件好事情！



这个HomeTheaterFacade帮助客户管理全部子系统组件，它的存在让客户变得简单又有弹性。

我们可以在不影响客户的情况下升级家庭影院的这些组件。

我们试着让子系统也能遵守最少知识原则。如果变得太复杂，有太多朋友牵涉其中，那么我们可以增加更多的外观，将此子系统分成几个层次。



设计箱内的工具

你的工具箱开始变重了。本章加入了几个模式，让你可以改变接口，并降低客户和系统之间的耦合。

OO基础

OO原则

封装变化
多用组合，少用继承
针对接口编程，不针对实现编程
为交互对象之间的松耦合设计而努力
类应该对扩展开放，对修改关闭。
依赖抽象，不要依赖具体类。
只和朋友交谈

我们有一个新的技巧，用来维护设计中的低层解耦（嘘！不要告诉太多人，只告诉你的朋友）……

OO模式

工厂方法模式——定义了一个创建对象
单件模式——确保一个类只有一个实
将请求封装成对象，
用不同的请求、队
请求来参数化其他

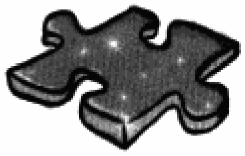
……还增加了两个新的模式。它们都会改变接口，适配器的意图是要转换接口，而外观的意图是要统一和简化接口。

适配器模式——将一个类的接口，转换成客户期望另一个接口。适配器让原本不兼容的类可以合作无间。

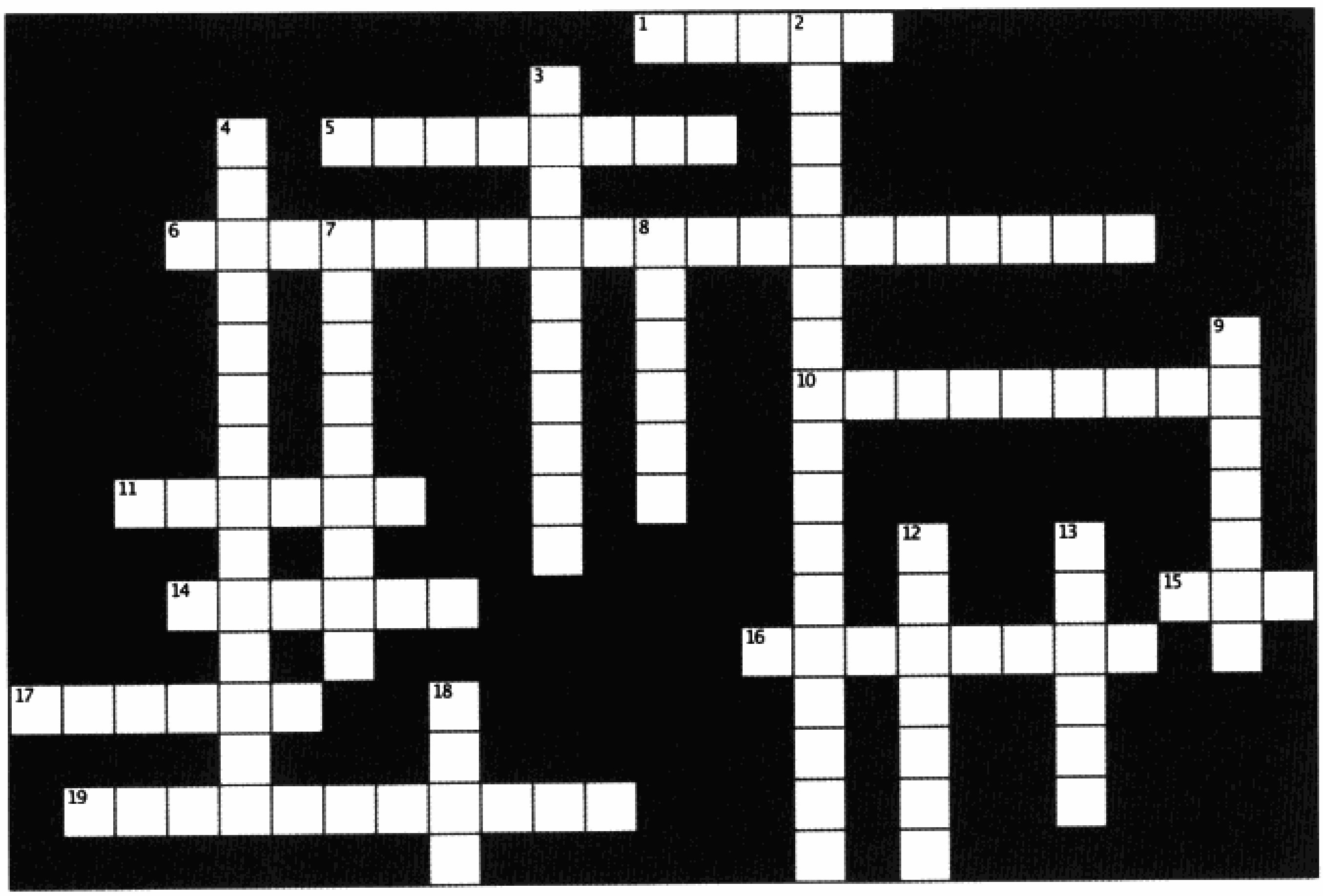
外观模式——提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

要点

- 当需要使用一个现有的类而其接口并不符合你的需要时，就使用适配器。
- 当需要简化并统一一个很大的接口或者一群复杂的接口时，使用外观。
- 适配器改变接口以符合客户的期望。
- 外观将客户从一个复杂的子系统中解耦。
- 实现一个适配器可能需要一番功夫，也可能不费功夫，视目标接口的大小与复杂度而定。
- 实现一个外观，需要将子系统组合进外观中，然后将工作委托给子系统执行。
- 适配器模式有两种形式：对象适配器和类适配器。类适配器需要用到多重继承。
- 你可以为一个子系统实现一个以上的外观。
- 适配器将一个对象包装起来以改变其接口；装饰者将一个对象包装起来以增加新的行为和责任；而外观将一群对象“包装”起来以简化其接口。



是的，又是拼字时间了。这些字都是来自本章的英文词汇。

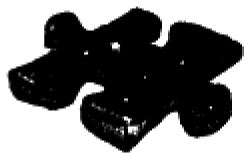


横排提示：

1. True or false, Adapters can only wrap one object
5. An Adapter _____ an interface
6. Movie we watched (5 words)
10. If in Europe you might need one of these (two words)
11. Adapter with two roles (two words)
14. Facade still _____ low level access
15. Ducks do it better than Turkeys
16. Disadvantage of the Principle of Least Knowledge: too many __________
17. A _________ simplifies an interface
19. New American dream (two words)

竖排提示：

2. Decorator called Adapter this (3 words)
3. One advantage of Facade
4. Principle that wasn't as easy as it sounded (two words)
7. A ______ adds new behavior
8. Masquerading as a Duck
9. Example that violates the Principle of Least Knowledge: System.out.______
12. No movie is complete without this
13. Adapter client uses the _________ interface
18. An Adapter and a Decorator can be said to _____ an object



习题解答

Sharpen your pencil

如果我们也需要一个将鸭子转换成火鸡的适配器，我们称它为DuckAdapter。请写下这个类：

我们将鸭子适配到火鸡，所以实现了火鸡接口。

```
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }

    public void gobble() {
        duck.quack();
    }

    public void fly() {
        if (rand.nextInt(5)  == 0) {
            duck.fly();
        }
    }
}
```

将我们使用到的鸭子记录下来。

我们也创建一个随机数对象，等一下在fly()中会用到。

把咯咯叫变成呱呱叫。

因为鸭子比火鸡更会飞，所以我们决定让鸭子平均五次只飞一次。

Sharpen your pencil

这些类有没有违反最少知识原则？请说明原因。

```
public House {
    WeatherStation station;

    // 其他的方法和构造器
    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}
```

违反最少知识原则了，因为在此调用的方法属于另一次调用的返回对象。

```
public House {
    WeatherStation station;

    // 其他的方法和构造器

    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

没有违反最少知识原则，但是，把程序改成这样真的有意义吗？

习题解答

你已经知道如何实现一个适配器，将Enumeration适配成Iterator。现在请你实现一个适配器，将Iterator适配成Enumeration。

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

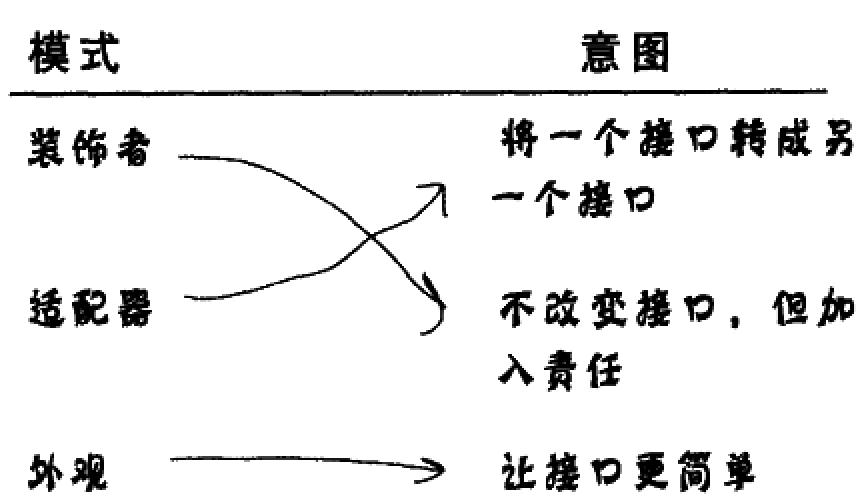
    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

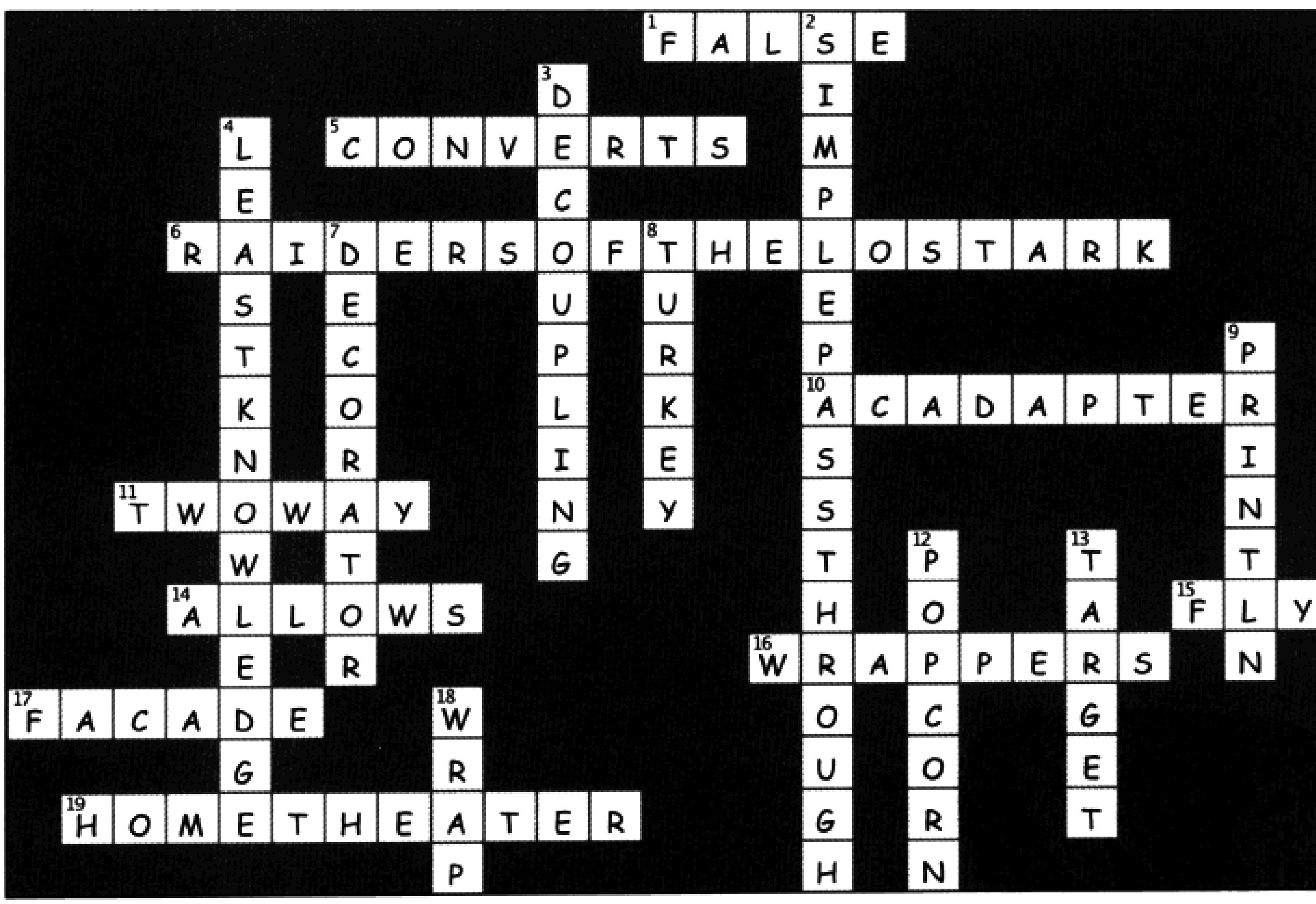
    public Object nextElement() {
        return iterator.next();
    }
}
```

连连看

找出每个模式的意图：



习题解答


1 FALSE
2 SIMPLEPASSTHROUGH
3 DECOUPLING
4 LEASTKNOWLEDG
5 CONVERTS
6 RAIDERSOFTHELOSTARK
7 DECORATOR
8 TURKEY
9 PRINTLN
10 ACADAPTER
11 TWOWAY
12 POPCORN
13 TARGET
14 ALLOWS
15 FLY
16 WRAPPERS
17 FACADE
18 WRAP
19 HOMETHEATER

8 模板方法模式

封装算法

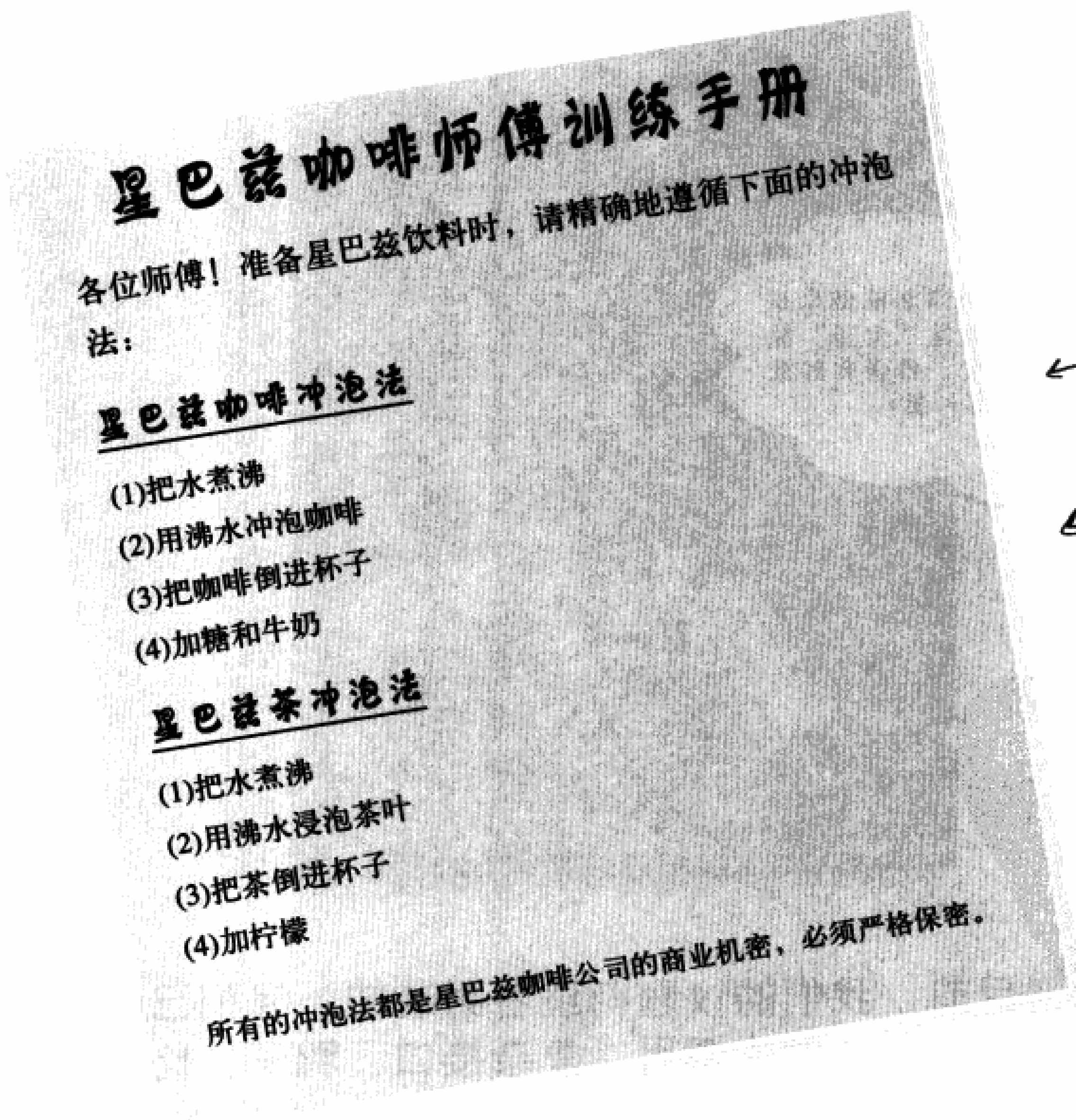


直到目前，我们的议题都绕着封装转；我们已经封装了对象创建、方法调用、复杂接口、鸭子、比萨……接下来呢？我们将要深入封装算法块，好让子类可以在任何时候都可以将自己挂接进运算里。我们甚至会在本章学到一个受到好莱坞影响而启发的设计原则。

多来点咖啡因吧

有些人没有咖啡就活不下去；有些人则离不开茶。两者共同的成分是什么？当然是咖啡因了！

但还不只这样；茶和咖啡的冲泡方式非常相似，不信你瞧瞧：



星巴兹咖啡师傅训练手册

各位师傅！准备星巴兹饮料时，请精确地遵循下面的冲泡法：

星巴兹咖啡冲泡法

(1)把水煮沸
(2)用沸水冲泡咖啡
(3)把咖啡倒进杯子
(4)加糖和牛奶

星巴兹茶冲泡法

(1)把水煮沸
(2)用沸水浸泡茶叶
(3)把茶倒进杯子
(4)加柠檬

所有的冲泡法都是星巴兹咖啡公司的商业机密，必须严格保密。

咖啡和茶的冲泡法大致上一样，不是吗？

快速搞定几个咖啡和茶的类（用Java语言）



让我们扮演“代码师傅”，写一些代码来创建咖啡和茶。

下面是咖啡：

这是我们的咖啡类，用来煮咖啡。

这是我们的咖啡冲泡法，直接取自训练手册。

每个步骤都被实现在分离的方法中。

这里每个方法都实现了算法中的一个步骤：煮沸水、冲泡咖啡、把咖啡倒进杯子、加糖和奶。

```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

接下来是茶……



```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

这看起来和前一页咖啡的实现很像，其中第2和第4个步骤不一样，但基本上是相同的冲泡法。

这两个方法是泡茶专有的。

请注意，这两个方法和咖啡的方法完全一样！也就是说，在这里出现了重复的代码。

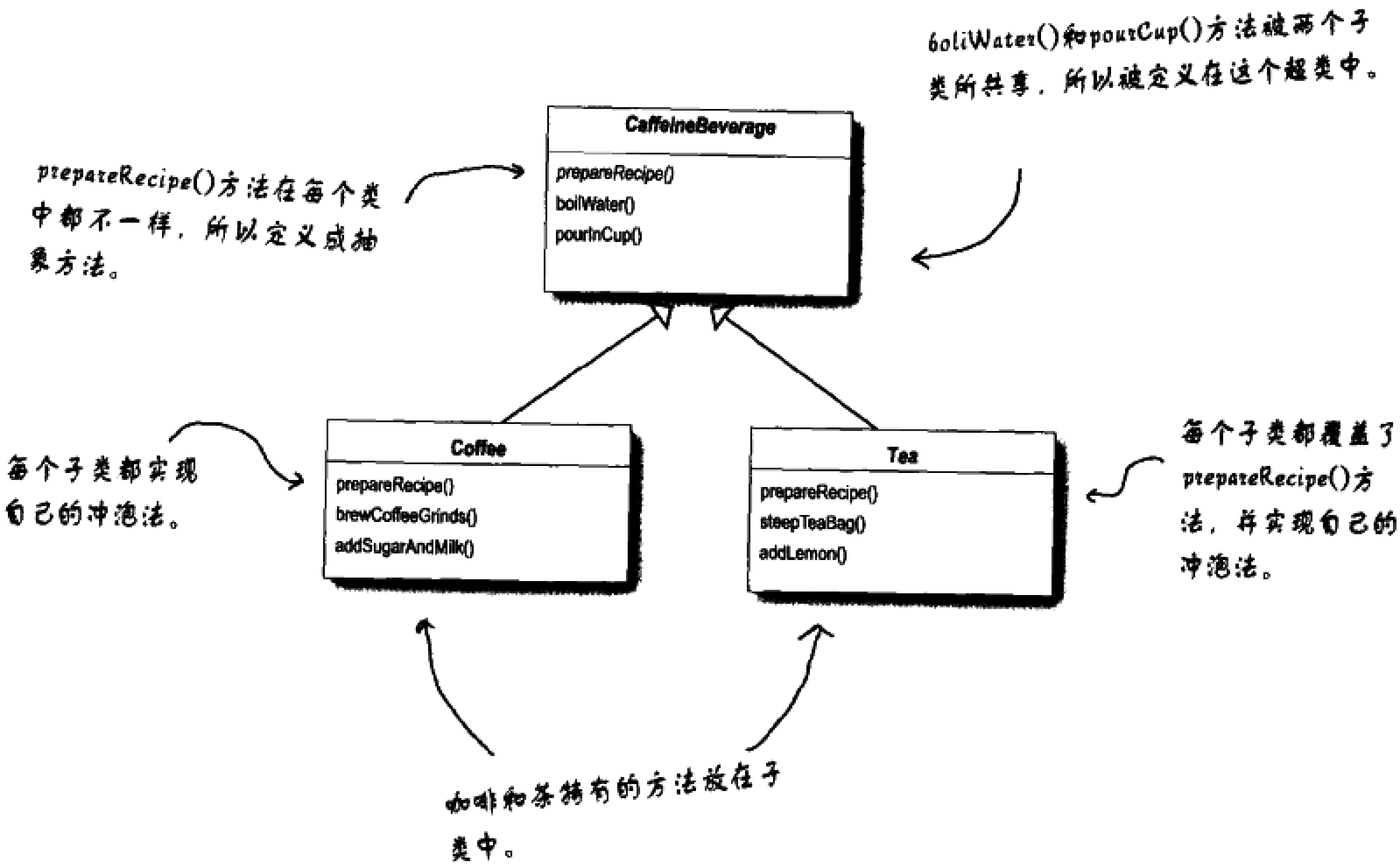


设计谜题

你已经看到茶和咖啡的类存在着重复的代码。请研究茶和咖啡的类，然后绘制一个类图，表达出你会如何重新设计这些类来删除重复代码：

先生，我能够抽取你的咖啡和茶吗？

看起来这个咖啡和茶类的设计练习相当直接。你的第一版设计，可能看起来像这样：



BRAIN POWER

我们的新设计你觉得怎样？嗯，再看一眼。我们是不是忽略了某些其他的共同点？咖啡和茶之间还有什么是相似的？

更进一步的设计……

所以，咖啡和茶还有什么其他的共同点呢？让我们先从冲泡法下手。



注意两份冲泡法都采用了相同的算法：

1. 把水煮沸。
2. 用热水泡咖啡或茶。
3. 把饮料倒进杯子。
4. 在饮料内加入适当的调料。

这两个并没有被抽出来，但是他们是一样的，只是应用在不同的饮料上。

这两个已经被抽出来，放到基类中了。

那么，我们有办法将prepareRecipe()也抽象化吗？是的，现在就来看看该怎么做……

抽象prepareRecipe()

让我们从每一个子类（也就是咖啡和茶）中逐步抽象prepareRecipe()……

❶ 我们所遇到的第一个问题，就是咖啡使用brewCoffeeGrinds()和addSugarAndMilk()方法，而茶使用steepTeaBag()和addLemon()方法。

咖啡

```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

茶

```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

让我们来思考这一点：浸泡（steep）和冲泡（brew）差异其实不大。所以我们给它一个新的方法名称，比方说brew()，然后不管是泡茶或冲泡咖啡我们都用这个名称。类似地，加糖和牛奶也和加柠檬很相似：都是在饮料中加入调料。让我们也给它一个新的方法名称来解决这个问题，就叫做addCondiments()好了。这样一来，新的prepareRecipe()方法看起来就像这样：

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

❷ 现在我们有了新的prepareRecipe()方法，但是需要让它能够符合代码。要想这么做，我们先从CaffeineBeverage（咖啡因饮料）超类开始：

咖啡因饮料是一个抽象类。

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

现在，用同一个prepareRecipe()方法来处理茶和咖啡。prepareRecipe()被声明为final，因为我们不希望子类覆盖这个方法！我们将步骤2和步骤4泛化成为brew()和addCondiments()。

因为咖啡和茶处理这些方法的做法不同，所以这两个方法必须被声明为抽象，剩余的东西留给子类去操心。

别忘了，我们将这些移到咖啡因饮料类中（回到我们的类图）。

3. 最后，我们需要处理咖啡和茶类了。这两个类现在都是依赖超类（咖啡因饮料）来处理冲泡法，所以只需要自行处理冲泡和添加调料部分：

茶和咖啡都是继承自咖啡因饮料。

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

茶需要定义brew()和addCondiments()，这两个抽象方法来自Beverage类。

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

咖啡也是一样，除了处理对象是咖啡，而调料是用糖和牛奶来取代茶包和柠檬。

Sharpen your pencil

绘制出新类图。现在，我们已经将prepareRecipe()的实现放在CaffeineBeverage类中了：

我们做了什么？

我们已经明白了两种冲泡法是基本相同的，只是一些步骤需要不同的实现。所以我们泛化了冲泡法，把它放在基类。

茶

1. 把水煮沸
2. 用沸水浸泡茶叶
3. 把茶倒进杯子
4. 加柠檬

咖啡

1. 把水煮沸
2. 用沸水冲泡咖啡粉
3. 把咖啡倒进杯子
4. 加糖和牛奶

泛化

泛化

咖啡因饮料

1. 把水煮沸
2. 冲泡
3. 把饮料倒进杯子
4. 加调料

一些步骤依赖子类进行

一些步骤依赖子类进行

茶子类



2. 用沸水浸泡茶叶
4. 加柠檬

咖啡子类



2. 用沸水冲泡咖啡
4. 加糖和牛奶

咖啡因饮料了解和控制冲泡法的步骤，亲自执行步骤1和步骤3，但依赖茶和咖啡来完成步骤2和步骤4。

认识模板方法

基本上，我们刚刚实现的就是模板方法模式。这是什么？让我们看看咖啡因饮料类的结构，它包含了实际的“模板方法”：

```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // 实现
    }

    void pourInCup() {
        // 实现
    }

}
```

prepareRecipe()是我们的模板方法。为什么？

因为：

(1) 毕竟它是一个方法。

(2)它用作一个算法的模板，在这个例子中，算法是用来制作咖啡因饮料的。

在这个模板中，算法内的每一个步骤都被一个方法代表了。

某些方法是由这个类（也就是超类）处理的……

……某些方法则是由子类处理的。

需要由子类提供的方法，必须在超类中声明为抽象。

模板方法定义了一个算法的步骤，并允许子类为一个或多个步骤提供实现。

走，泡茶去……



让我们逐步地泡茶，追踪这个模板方法是如何工作的。你会得知在算法内的某些地方，该模板方法控制了算法。它让子类能够提供某些步骤的实现……

```
boilWater();
brew();
pourInCup();
addCondiments();
```

prepareRecipe方法控制了算法，没有人能够改变它。这个方法也会依赖子类来提供某些或所有步骤的实现。

❶ 好吧！首先我们需要一个茶对象……

```
Tea myTea = new Tea();
```

❷ 然后我们调用这个模板方法：

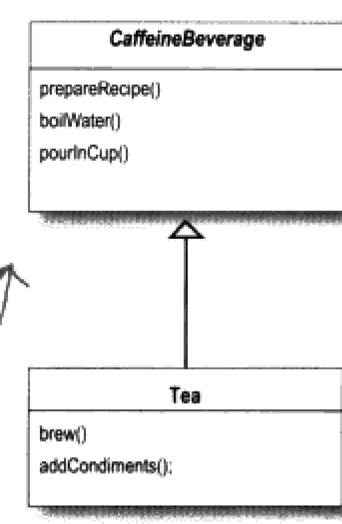
```
myTea.prepareRecipe();
```

它会依照算法来制作咖啡因饮料……

❸ 首先，把水煮沸：

```
boilWater();
```

这件事情是在咖啡因饮料类（超类）中进行的。



❹ 接下来，我们需要泡茶，这件事情只有子类才知道要怎么做：

```
brew();
```

❺ 现在把茶倒进杯子中；所有的饮料做法都一样，所以这件事情发生在超类中：

```
pourInCup();
```

❻ 最后，我们加进调料，由于调料是各个饮料独有的，所以由子类来实现它：

```
addCondiments();
```

模板方法带给我们什么？





不好的茶和咖啡实现	模板方法提供的酷炫咖啡因饮料
Coffee和Tea主导一切；它们控制了算法。	由CaffeineBeverage类主导一切，它拥有算法，而且保护这个算法。
Coffee和Tea之间存在着重复的代码。	对子类来说，CaffeineBeverage类的存在，可以将代码的复用最大化。
对于算法所做的代码改变，需要打开子类修改许多地方。	算法只存在于一个地方，所以容易修改。
由于类的组织方式不具有弹性，所以加入新种类的咖啡因饮料需要做许多工作。	这个模板方法提供了一个框架，可以让其他的咖啡因饮料插进来。新的咖啡因饮料只需要实现自己的方法就可以了。
算法的知识和它的实现会分散在许多类中。	CaffeineBeverage类专注在算法本身，而由子类提供完整的实现。

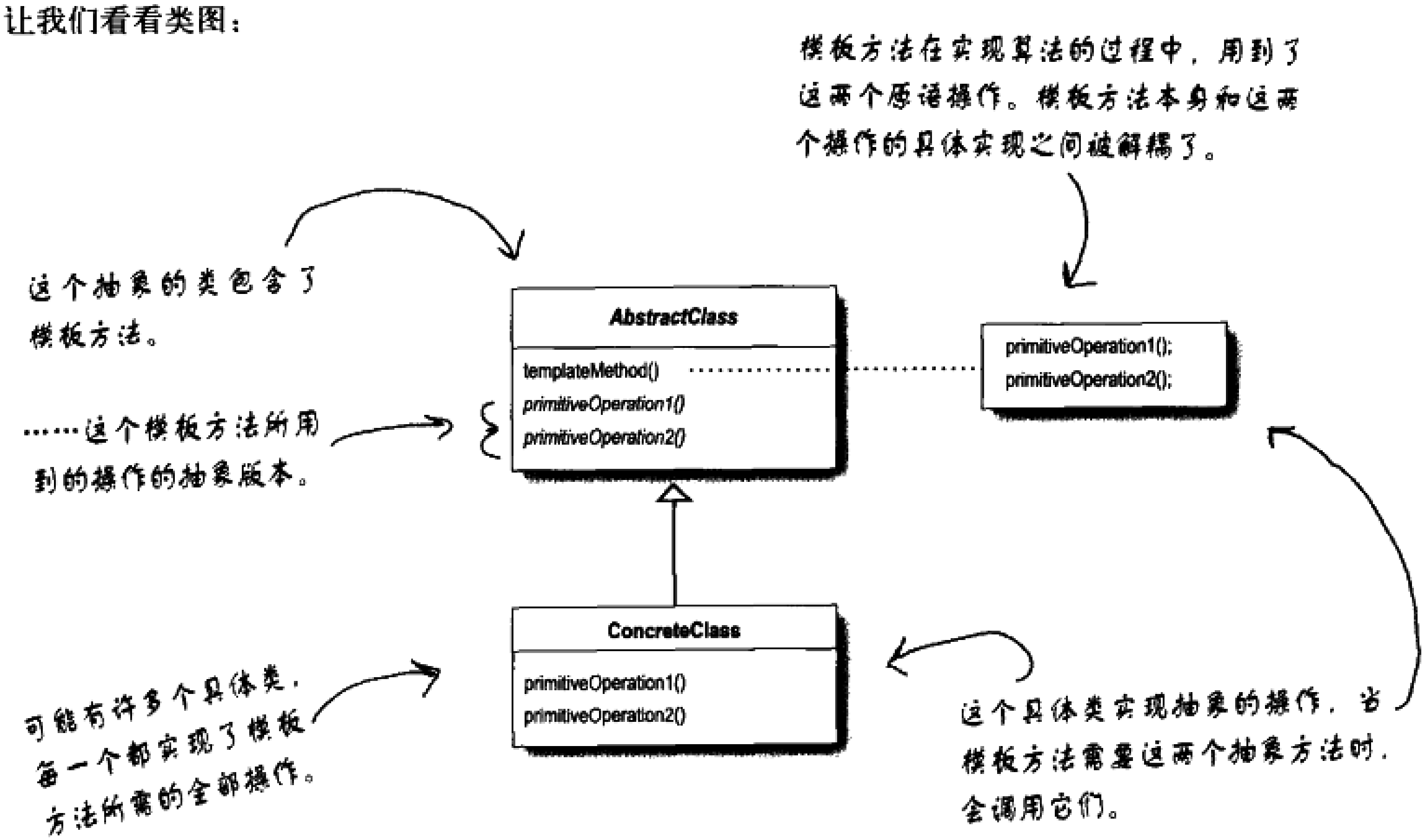
定义模板方法模式

你已经看到了在茶和咖啡的例子中如何使用模板方法模式。现在，就让我们来看看这个模式的正式定义和所有的细节：

> **模板方法模式**在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

这个模式是用来创建一个算法的模板。什么是模板？如你所见的，模板就是一个方法。更具体地说，这个方法将算法定义成一组步骤，其中的任何步骤都可以是抽象的，由子类负责实现。这可以确保算法的结构保持不变，同时由子类提供部分实现。

让我们看看类图：



再靠近一点

让我们细看抽象类是如何被定义的，包括了它内含的模板方法和原语操作。

这就是我们的抽象类。它被声明为抽象，用来作为基类，其子类必须实现其操作。

这就是模板方法。它被声明为final，以免子类改变这个算法的顺序。

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // 这里是实现
    }
}
```

模板方法定义了一连串的步骤，每个步骤由一个方法代表。

在这个范例中有两个原语操作，具体子类必须实现它们。

这个抽象类有一个具体的操作。关于这类方法，稍后会再详述……

更靠近一点

现在我们要“更靠近一点”，详细看看此抽象类内可以有哪些类型的方法：

我们加进一个新方法调用，改变了tempateMethod()。

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // 这里是实现
    }

    void hook() {}

}
```

这两个方法还是和以前一样，定义成抽象，由具体的子类实现。

这个具体的方法被定义在抽象类中。将它声明为final，这样一来子类就无法覆盖它。它可以被模板方法直接使用，或者被子类使用。

这是一个具体的方法，但它什么事情都不做！

我们也可以有“默认不做事的方法”，我们称这种方法为“hook”（钩子）。子类可以视情况决定要不要覆盖它们。在下一页，我们就会知道钩子的实际用途。

对模板方法进行挂钩……



钩子是一种被声明在抽象类中的方法，但只有空的或者默认的实现。钩子的存在，可以让子类有能力对算法的不同点进行挂钩。要不要挂钩，由子类自行决定。

钩子有好几种用途，让我们先看其中一个，稍后再看其他几个：

```
public abstract class CaffeineBeverageWithHook {

    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

我们加上了一个小小的条件语句，而该条件是否成立，是由一个具体方法customerWantsCondiments()决定的。如果顾客“想要”调料，只有这时我们才调用addCondiments()。

我们在这里定义了一个方法，（通常）是空的缺省实现。这个方法只会返回true，不做别的事。

这就是一个钩子，子类可以覆盖这个方法，但不见得一定要这么做。

使用钩子

为了使用钩子，我们在子类中覆盖它。在这里，钩子控制了咖啡因饮料是否执行某部分算法；说得更明确一些，就是饮料中是否要加进调料。

我们如何得知顾客是否想要调料呢？开口问不就行了！

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {
        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

你覆盖了这个钩子，提供了自己的功能。

让用户输入他们对调料的决定。根据用户的输入，返回true或false。

这段代码询问用户是否想要奶和糖，通过命令行获得用户输入。

执行测试程序

好了，水开了……下面是一段测试码，用来制造热茶和热咖啡。

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

创建一杯茶。

一杯咖啡。

然后调用两者的prepareRecipe()。

执行结果……

```
File Edit Window Help send-more-honesttea
%java BeverageTestDrive

Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

一杯热腾腾的茶，是的，当然要加柠檬！

一杯热腾腾的咖啡，但是那些能让我腰围增大的有热量的调料就免了。



你知道吗？我们同意你的看法。但是你必须承认，这个例子实在很酷，钩子竟然能够作为条件控制，影响抽象类中的算法流程，实在很不赖吧！

我们相信，在你自己的代码中你一定可以找到其他真实的场面可以使用模板模式和钩子。

---

there are no Dumb Questions

问：当我创建一个模板方法时，怎么才能知道什么时候该使用抽象方法，什么时候使用钩子呢？

答：当你的子类"必须"提供算法中某个方法或步骤的实现时，就使用抽象方法。如果算法的这个部分是可选的，就用钩子。如果是钩子的话，子类可以选择实现这个钩子，但并不强制这么做。

问：使用钩子真正的目的是什么？

答：钩子有几种用法。如我们之前所说的，钩子可以让子类实现算法中可选的部分，或者在钩子对于子类的实现并不重要的时候，子类可以对此钩子置之不理。钩子的另一个用法，是让子类能够有机会对模板方法中某些即将发生的（或刚刚发生的）步骤作出反应。比方说，名为justReOrderedList()的钩子方法允许子类在内部列表重新组织后执行某些动作（例如在屏幕上重新显示数据）。正如你刚刚看到的，钩子也可以让子类有能力为其抽象类作一些决定。

问：子类必须实现抽象类中的所有方法吗？

答：是的，每一个具体的子类都必须定义所有的抽象方法，并为模板方法算法中未定义步骤提供完整的实现。

问：似乎我应该保持抽象方法的数目越少越好，否则，在子类中实现这些方法将会很麻烦。

答：当你在写模板方法的时候，心里要随时记得这一点。想要做到这一点，可以让算法内的步骤不要切割得太细，但是如果步骤太少的话，会比较没有弹性，所以要看情况折衷。

也请记住，某些步骤是可选的，所以你可以将这些步骤实现成钩子，而不是实现成抽象方法，这样就可以让抽象类的子类的负荷减轻。

好莱坞原则



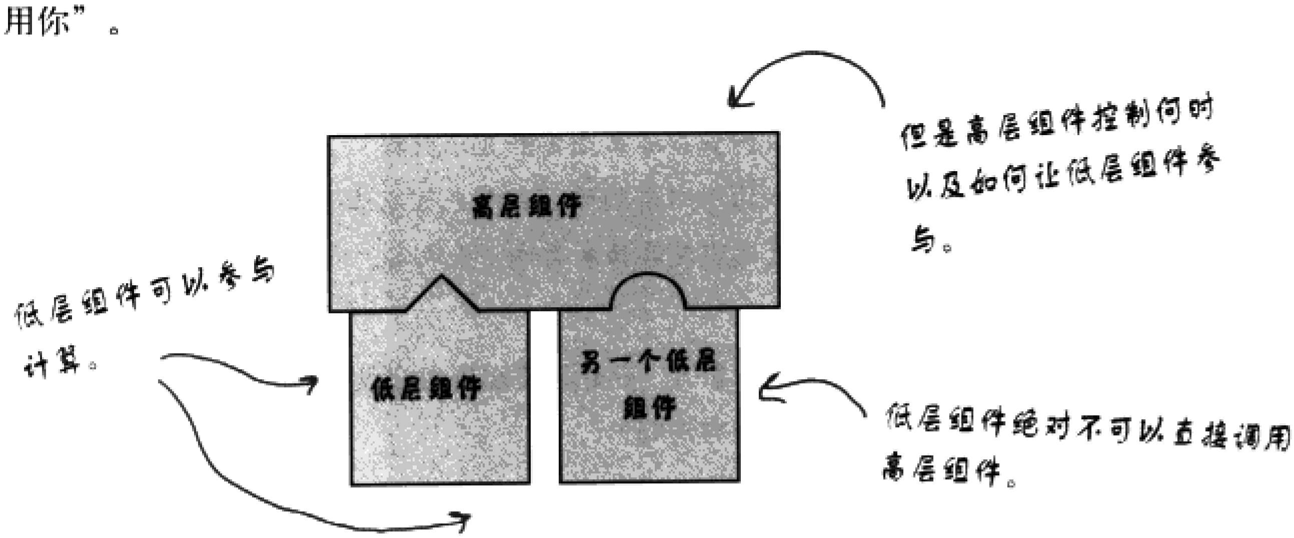
我们有一个新的设计原则，称为好莱坞原则：

> **好莱坞原则**
>
> **别调用（打电话给）我们，我们会调用（打电话给）你。**

很容易记吧？但这和OO设计又有什么关系呢？

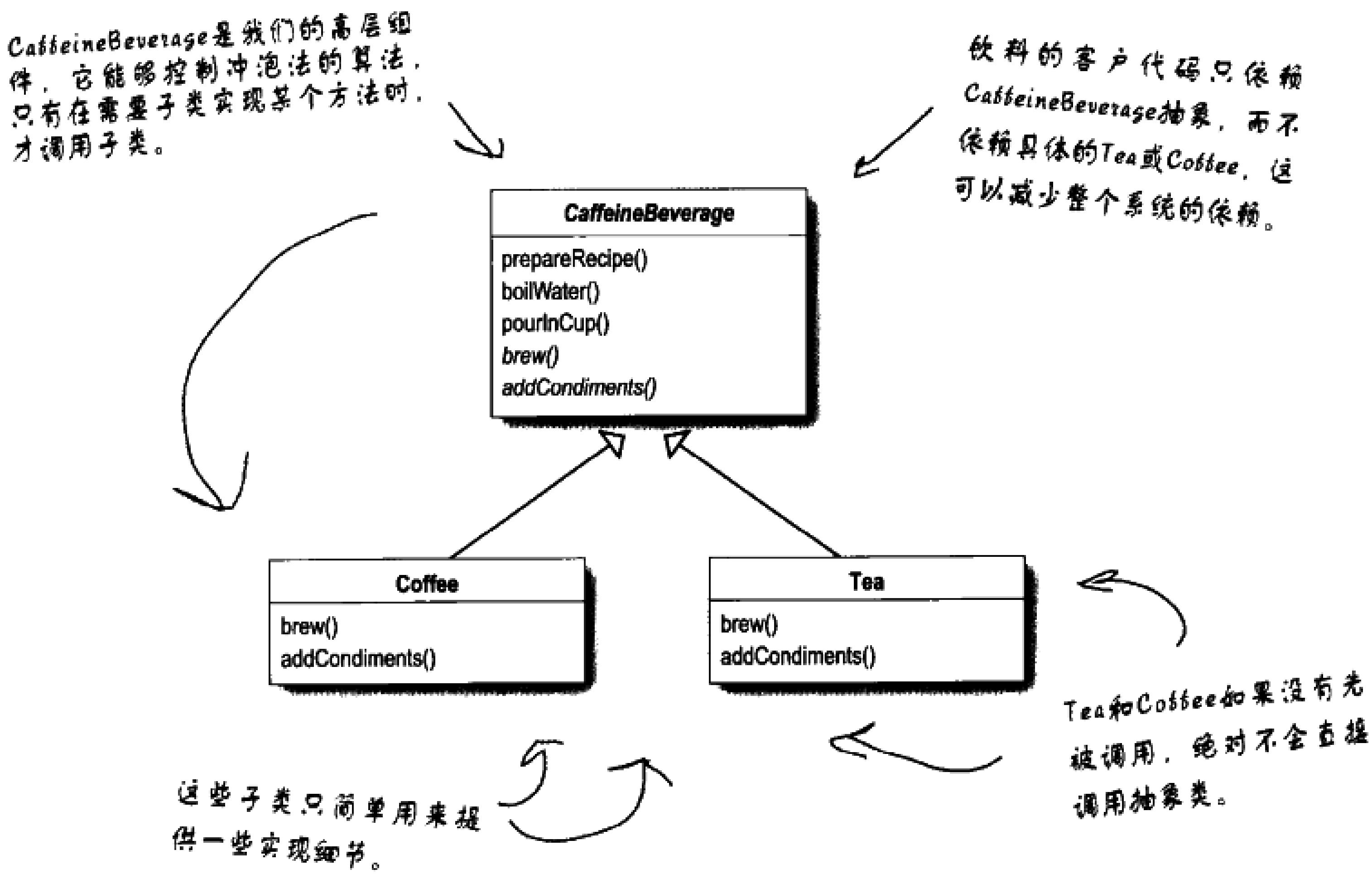
好莱坞原则可以给我们一种防止“依赖腐败”的方法。当高层组件依赖低层组件，而低层组件又依赖高层组件，而高层组件又依赖边侧组件，而边侧组件又依赖低层组件时，依赖腐败就发生了。在这种情况下，没有人可以轻易地搞懂系统是如何设计的。

在好莱坞原则之下，我们允许低层组件将自己挂钩到系统上，但是高层组件会决定什么时候和怎样使用这些低层组件。换句话说，高层组件对待低层组件的方式是“别调用我们，我们会调用你”。



好莱坞原则和模板方法

好莱坞原则和模板方法之间的连接其实还算明显：当我们设计模板方法模式时，我们告诉子类，“不要调用我们，我们会调用你”。怎样才能办到呢？让我们再看一次咖啡因饮料的设计：



BRAIN POWER

还有哪些模式采用了好莱坞原则？

工厂方法、观察者；还有其他的模式吗？

there are no Dumb Questions

问：好莱坞原则和依赖倒置原则（第4章）之间的关系如何？

答： 依赖倒置原则教我们尽量避免使用具体类，而多使用抽象。而好莱坞原则是用在创建框架或组件上的一种技巧，好让低层组件能够被挂钩进计算中，而且又不会让高层组件依赖低层组件。两者的目标都是在于解耦，但是依赖倒置原则更加注重如何在设计中避免依赖。

好莱坞原则教我们一个技巧，创建一个有弹性的设计，允许低层结构能够互相操作，而又防止其他类太过依赖它们。

问：低层组件不可以调用高层组件中的方法吗？

答： 并不尽然。事实上，低层组件在结束时，常常会调用从超类中继承来的方法。我们所要做的是，避免让高层和低层组件之间有明显的环状依赖。

连连看

将模式和叙述之间配对：

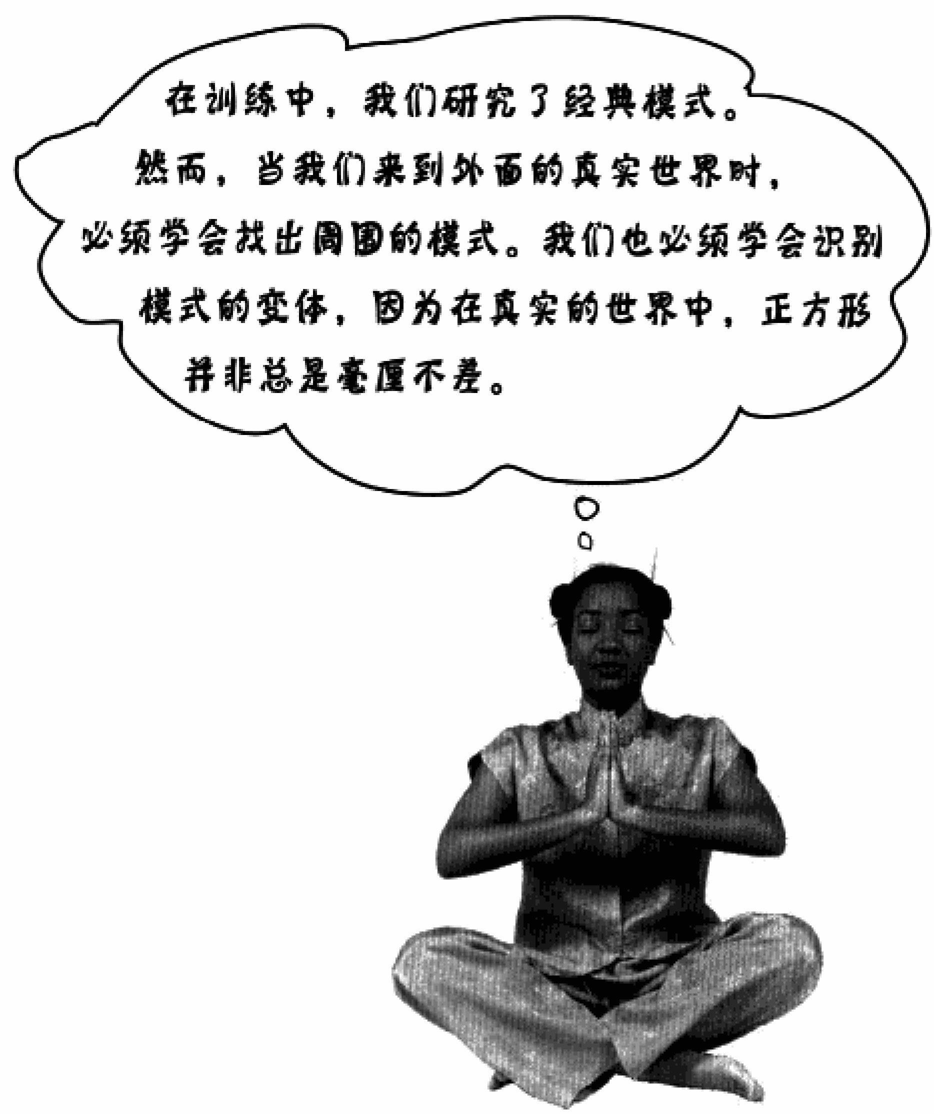
模式	叙述
模板方法	封装可互换的行为，然后使用委托来决定要采用哪一个行为
策略	子类决定如何实现算法中的步骤
工厂方法	由子类决定实例化哪个具体类

荒野中的模板方法

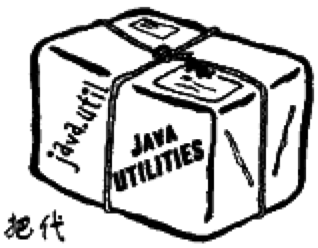
模板方法模式是一个很常见的模式，到处都是。尽管如此，你必须拥有一双锐利的眼睛，因为模板方法有许多实现，而它们看起来并不一定和书上所讲的设计一致。

这个模式很常见是因为对创建框架来说，这个模式简直棒极了。由框架控制如何做事情，而由你（使用这个框架的人）指定框架算法中每个步骤的细节。

让我们步入荒野，展开狩猎吧！（好啦！荒野就是指Java API）……



用模板方法排序



我们经常需要数组做什么事情？对了！排序。

Java数组类的设计者提供给我们一个方便的模板方法用来排序。让我们看看这个方法如何运行：

为了便于解释，我们把代码稍微简化了。如果你想看完整的代码，就去看Sun的源码吧……

这里其实有两个方法，共同提供排序的功能。

第一个方法sort()只是一个辅助（helper）方法，用来创建一个数组的拷贝，然后将其传递给mergeSort()方法当作目标数组。同时传入mergeSort()的参数，还包括数组的长度、以及从头（0）开始排序。

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

mergeSort()方法包含排序算法，此算法依赖于compareTo()方法的实现来完成算法。

把这想成是一个模板方法。

```
private static void mergeSort(Object src[], Object dest[],
            int low, int high, int off)
{

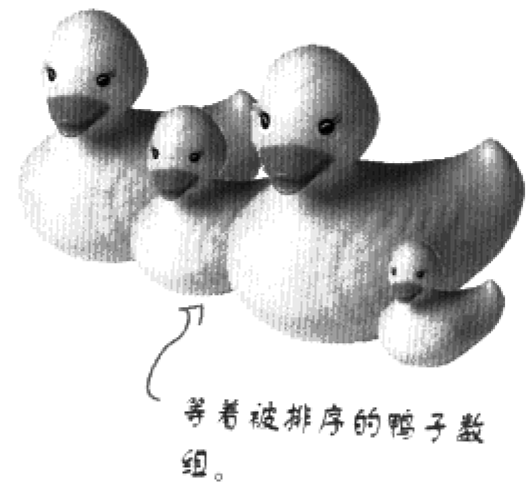
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
            ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    return;
}
```

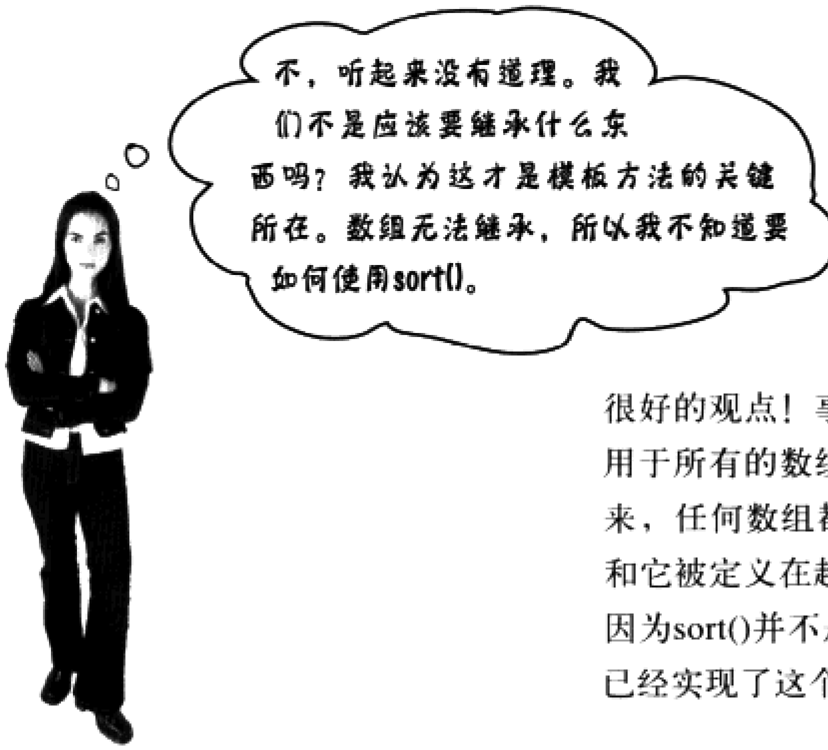
这是一个具体方法，已经在数组类中定义了。

我们需要实现compareTo()方法，"填补"模板方法的缺憾。

来排序鸭子吧……

假如我们有一个鸭子的数组需要排序，你要怎么做？数组的排序模板方法已经提供了算法，但是你必须让这个模板方法知道如何比较鸭子。你所要做的事情就是，实现一个compareTo()方法……听起来有道理吧？



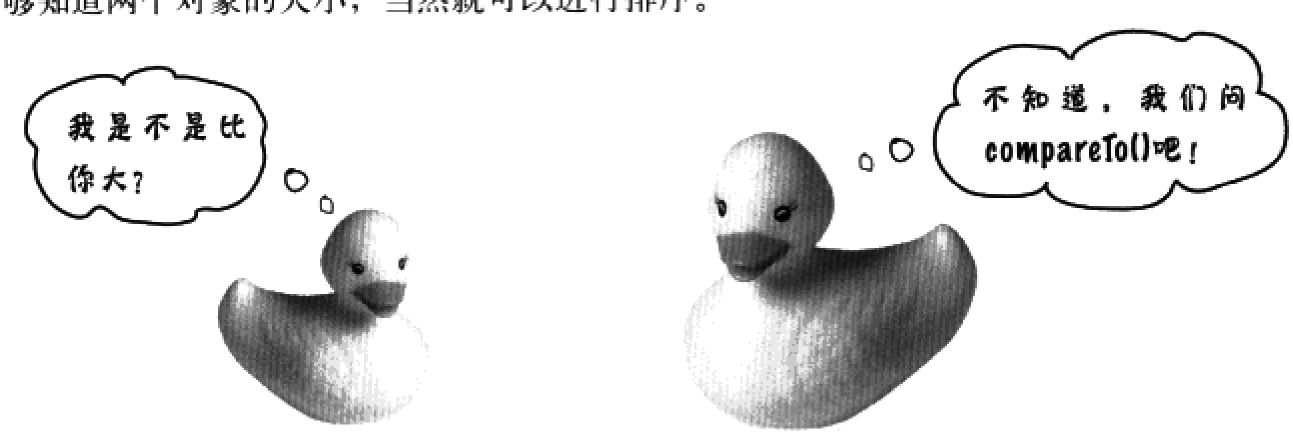


很好的观点！事情是这样的：sort()的设计者希望这个方法能使用于所有的数组，所以他们把sort()变成是静态的方法，这样一来，任何数组都可以使用这个方法。但是没关系，它使用起来和它被定义在超类中是一样的。现在，还有一个细节要告诉你：因为sort()并不是真正定义在超类中，所以sort()方法需要知道你已经实现了这个compartTo()方法，否则就无法进行排序。

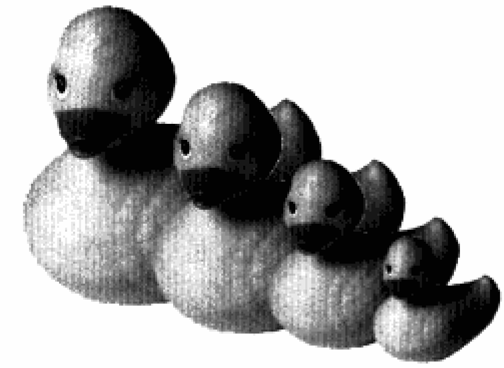
要达到这一点，设计者利用了Comparable接口。你须实现这个接口，提供这个接口所声明的方法，也就是compareTo()。

什么是compareTo()？

这个compareTo()方法将比较两个对象，然后返回其中一个是大于、等于还是小于另一个。sort()只要能够知道两个对象的大小，当然就可以进行排序。



比较鸭子



好了，现在你知道了如果要排序鸭子，就必须实现这个compareTo()方法；然后，数组就可以被正常地排序了。

鸭子的实现如下：

请记住，我们需要让鸭子类实现Comparable接口，因为我们无法真的让鸭子数组去继承数组。

```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }

    public int compareTo(Object object) {

        Duck otherDuck = (Duck)object;

        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

鸭子有名字和体重。

尽量让这里简单：只打印出名字和体重。

好了，这就是排序所要用的……

compareTo()需要被传入另一只鸭子，和本身这只鸭子做比较。

我们在这里指定鸭子是如何比较的。如果这只鸭子的体重比另一只鸭子的体重轻，就返回-1；如果相等，就返回0；如果这只鸭子的体重较重，就返回1。

让我们排序一些鸭子

这是测试排序鸭子的程序……

```
public class DuckSortTestDrive {
    public static void main(String[] args) {
        Duck[] ducks = {
                        new Duck("Daffy", 8),
                        new Duck("Dewey", 2),
                        new Duck("Howard", 7),
                        new Duck("Louie", 2),
                        new Duck("Donald", 10),
                        new Duck("Huey", 2)
         };

        System.out.println("Before sorting:");
        display(ducks);

        Arrays.sort(ducks);

        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (int i = 0; i < ducks.length; i++) {
            System.out.println(ducks[i]);
        }
    }
}
```

我们需要一个鸭子数组；这些看起来不错。

请注意，我们调用Array类的静态方法sort()，然后将鸭子数组当作参数传入。

将它们打印出来，看鸭子们的名字和体重。

开始排序了！

再将它们打印出来，看鸭子们的名字和体重。

执行结果！

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive

Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
```

还没排序的鸭子

已经排序的鸭子

观察鸭子排序的内部工作

让我们追踪Array类的sort()模板方法的工作过程。我们会看到模板方法是如何控制算法的，以及在算法中的某些点上它是如何要求我们的鸭子提供某个步骤的实现的……

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

sort()方法控制算法，没有类可以改变这一点。sort()依赖一个Comparable类提供compareTo()的实现。

1. 首先，我们需要一个鸭子数组：

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

2. 然后调用Array类的sort()模板方法，并传入鸭子数组：

```
Arrays.sort(ducks);
```

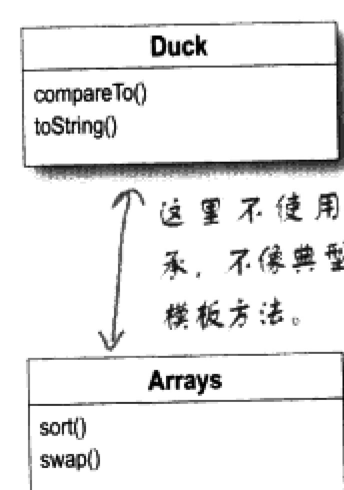
这个sort()方法（和它的helper mergeSort()）控制排序过程。

3. 想要排序一个数组，你需要一次又一次地比较两个项目，直到整个数组都排序完毕。

当比较两只鸭子的时候，排序方法需要依赖鸭子的compareTo()方法，以得知谁大谁小。第一只鸭子的compareTo()方法被调用，并传入另一只鸭子当成比较对象：

```
ducks[0].compareTo(ducks[1]);
```

第一只鸭子　　比较对象



这里不使用继承，不像典型的模板方法。

4. 如果鸭子的次序不对，就用Array的具体swap()方法将两者对调：

```
swap()
```

5. 排序方法会持续比较并对调鸭子，直到整个数组的次序是正确的！

there are no Dumb Questions

问：这真的是一个模板方法模式吗？还是你的想象力太丰富了？

答： 这个模式的重点在于提供一个算法，并让子类实现某些步骤而数组的排序做法很明显地并非如此！但是，我们都知道，荒野中的模式并非总是如同教科书例子一般地中规中矩，为了符合当前的环境和实现的约束，它们总是要被适当地修改。

这个Array类sort()方法的设计者受到一些约束。通常我们无法设计一个类继承Java数组，而sort()方法希望能够适用于所有的数组（每个数组都是不同的类）。所以它们定义了一个静态方法，而由被排序的对象内的每个元素自行提供比较大小的算法部分。

所以，这虽然不是教科书上的模板方法，但它的实现仍然符合模板方法模式的精神。再者，由于不需要继承数组就可以使用这个算法，这样使得排序变得更有弹性、更有用。

问：排序的实现实际上看起来更像是策略模式，而不是模板方法模式。为什么我们要将它归为模板方法？

答： 你之所以会这么认为，可能是因为策略模式使用对象组合。在某种程度上，你是对的——我们使用数组对象排序我们的数组，这部分和策略模式非常相似。但是请记住，在策略模式中，你所组合的类实现了整个算法。数组所实现的排序算法并不完整，它需要一个类填补compareTo()方法的实现。因此，我们认为这更像模板方法。

问：在Java API中，还有其他模板方法的例子吗？

答： 是的，你可以在一些地方看到它们。比方说，java.io的InputStream类有一个read()方法，是由子类实现的，而这个方法又会被read(byte b[], int off,int len)模板方法使用。

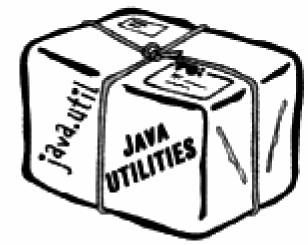
BRAIN POWER

我们知道应该多用组合，少用继承，对吧？sort()模板方法的实现决定不使用继承，sort方法被实现成一个静态的方法，在运行时和Comparable组合。这样的做法有何优缺点？你如何处置这个难题？难道Java数组让这一切变得特别麻烦吗？

BRAIN 2 POWER

想一想另一个模式，它是模板方法的一种特殊状况，原语操作用来创建并返回对象。这是什么模式？

写一个Swing的窗口程序



在我们模板方法的狩猎历程中，你要特别注意Swing的JFrame！

也许你没用过JFrame，在这里简单解释一下。它是最基本的Swing容器，继承了一个paint()方法。在默认状态下，paint()是不做事情的，因为它是一个"钩子"！通过覆盖paint()，你可以将自己的代码插入JFrame的算法中，显示出你所想要的画面。下面是一个超级简单的例子：

我们扩展了JFrame，它包含一个update()方法，这个方法控制更新屏幕的算法。我们可以通过覆盖paint()钩子方法和这个算法挂上钩。

不用管里面的细节，这只是一些初始化的动作……

JFrame的更新算法被称为paint()。在默认状态下，paint()是不做事的……它只是一个钩子。我们覆盖paint()，告诉JFrame在窗口上面画出一条消息。

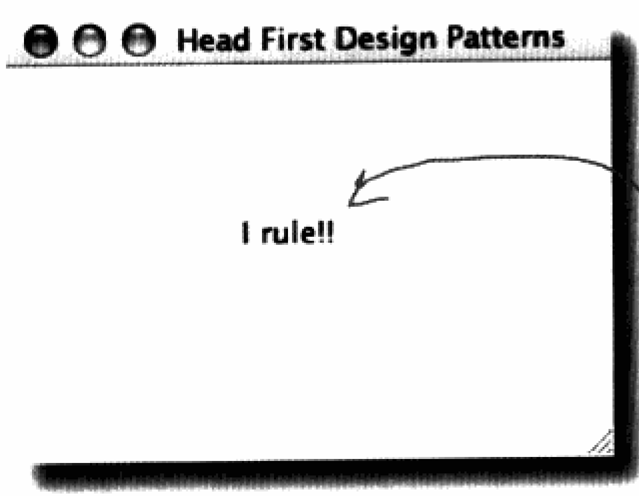
```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

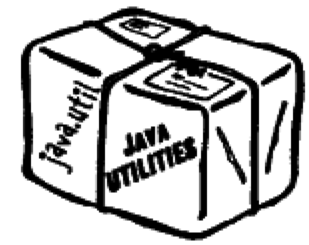
    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```



因为我们利用了paint()钩子方法，所以可以显示出这样的消息。

Applet



我们最后的狩猎目标：applet。

你大概知道applet就是一个能够在网页上面执行的小程序。任何applet都必须继承自Applet类，而Applet类中提供了好些钩子，让我们来看看其中的几个：

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // applet正在被销毁……
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

init钩子用来进行applet的初始化动作，它会在applet一开始的时候被调用一次。

repaint()是Applet类的一个具体方法，可让applet的上层组件知道这个applet需要重绘。

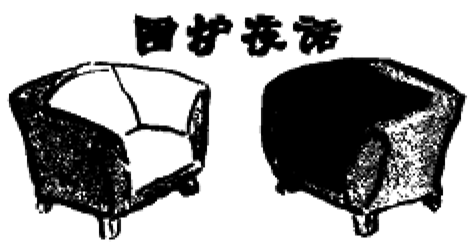
这个start钩子可以在applet正要被显示在网页上时，让applet做一些动作。

如果用户跳到别的网页，这个stop钩子会被调用，然后applet就可以在这里做一些事情来停止它的行动。

当这个applet即将被销毁（例如：关闭浏览器）时，destroy钩子就会被调用。我们可以在这里显示一些东西，但这么做好像没什么意义？

看看是谁在这里呀！这不正是我们的老朋友paint()方法吗？applet也将这个方法当做钩子了。

具体的applet大量使用钩子来提供行为。因为这些行为是作为钩子实现的，所以Applet类就不用去实现它们。

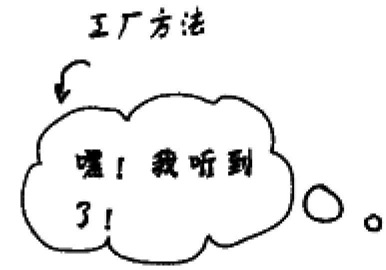


今夜话题：模板方法和策略的比较。

模板方法

策略您好，您怎么会出现在我的章节中呢？我还以为必须跟一些无聊的家伙，像是工厂方法，在一起呢！

策略



不，的确是我，不过你说话要小心——你和工厂方法不是有关联吗？

模板方法：我只是在开玩笑啦！说正经的，你在这儿干什么呢？我们足足有八个章节没有看到你了！

策略：我听说你的章节已经接近尾声，所以特地来看看事情怎么样。我们有许多共同点，所以我想或许可以提供一些帮助……

模板方法：你可能得再向读者自我介绍一下，因为你已经消失很久了。

策略：不见得如此。从第1章开始，我在逛街的时候，老是被路人拦了下来，他们说“你不是那个什么模式来着……”，所以，我想他们知道我是谁。不过为了你，我再说一次好了：我定义一个算法家族，并让这些算法可以互换。正因为每一个算法都被封装起来了，所以客户可以轻易地使用不同的算法。

模板方法：嘿！听起来好像是我在做的事情。但是我的意图和你有点不太一样：我的工作是要定义一个算法的大纲，而由我的子类定义其中某些步骤的内容。这么一来，我在算法中的个别步骤可以有不同的实现细节，但是算法的结构依然维持不变。不过你就不一样了，似乎你必须放弃对算法的控制。

策略：我不确定话可以这么说……更何况，我并不是使用继承进行算法的实现，我是通过对象组合的方式，让客户可以选择算法实现。

模板方法

这我记得。但是我对算法有更多的控制权，而且不会重复代码。事实上，除了极少的一部分之外，我的算法的每一个部分都是相同的，所以我的类比你的有效率得多。会重复使用到的代码，都被我放进了超类中，好让所有的子类共享。

策略

你或许更有效率一点（只是一点点），也的确需要更少的对象。和我所采用的委托模型比起来，你也没那么复杂。但是因为我使用对象组合，所以我更有弹性。利用我，客户就可以在运行时改变他们的算法，而客户所需要做的，只是改用不同的策略对象罢了。拜托，作者选择把我摆在第1章，这不是没有道理的！

模板方法

好吧，我真替你感到高兴，但是你别忘了，环顾四周，我可是最常被使用的模式。为什么呢？因为我在超类中提供了一个基础的方法，达到代码的复用，并允许子类指定行为。我相信你会看到这一点在创建框架时是非常棒的！

策略

也许呢……但是，别忘了依赖！你的依赖程度比我高。

模板方法

这话怎么说？我的超类是抽象的。

策略

但是你必须依赖超类中的方法的实现，因为这是你算法中的一部分。但我就不同了，我不依赖任何人，整个算法我自己搞定！

模板方法

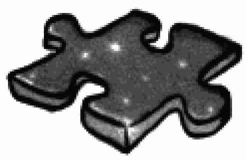
策略呀！就如同我所说的，我真为你感到高兴。谢谢你来拜访我，但我必须把这个章节剩下的部分完成。

策略

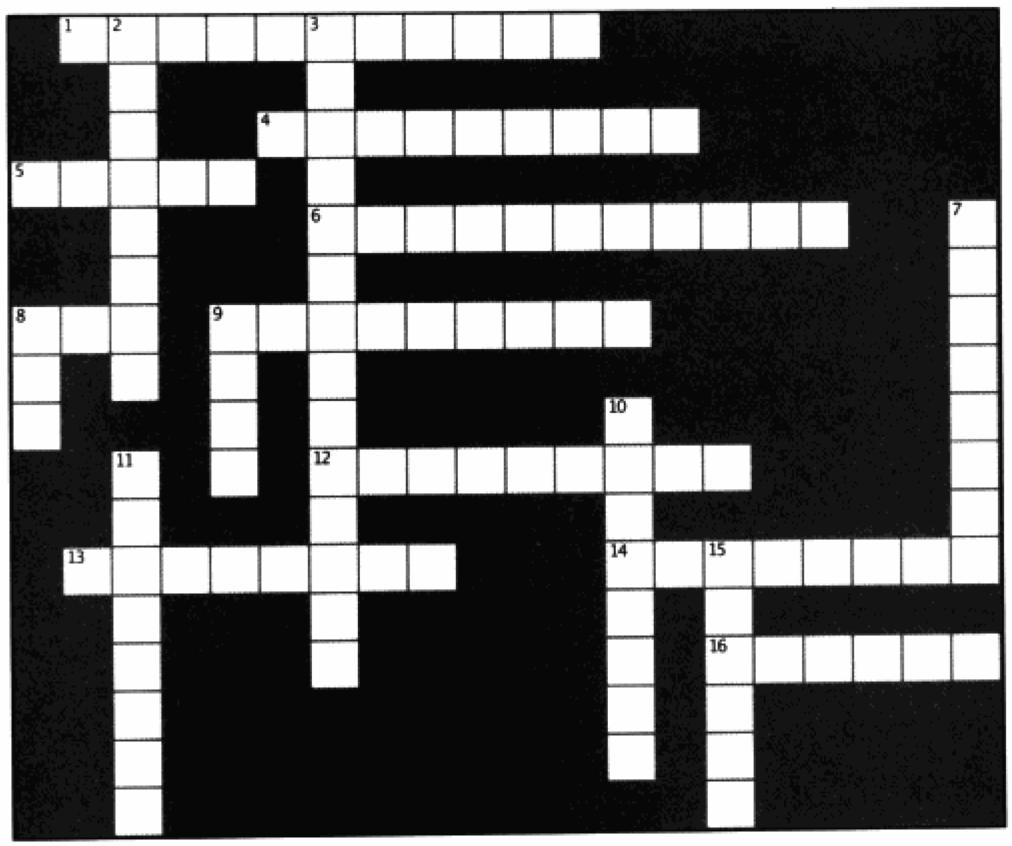
好啦！好啦！不要这么敏感。我让你继续工作吧，但是如果你需要我的特殊技能，请让我知道，我总是乐于助人的。

模板方法

知道了，别打电话给我，我会打电话给你……



又是这个时候了。

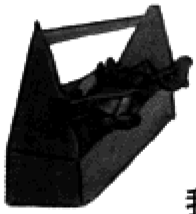


横排提示：

1. Strategy uses __________ rather than inheritance
4. Type of sort used in Arrays
5. The JFrame hook method that we overrode to print "I Rule"
6. The Template Method Pattern uses _____________ to defer implementation to other classes
8. Coffee and _____
9. Don't call us, we'll call you is known as the ___________________ Principle
12. A template method defines the steps of an _____________
13. In this chapter we gave you more _______________
14. The template method is usually defined in an ____________ class
16. Class that likes web pages

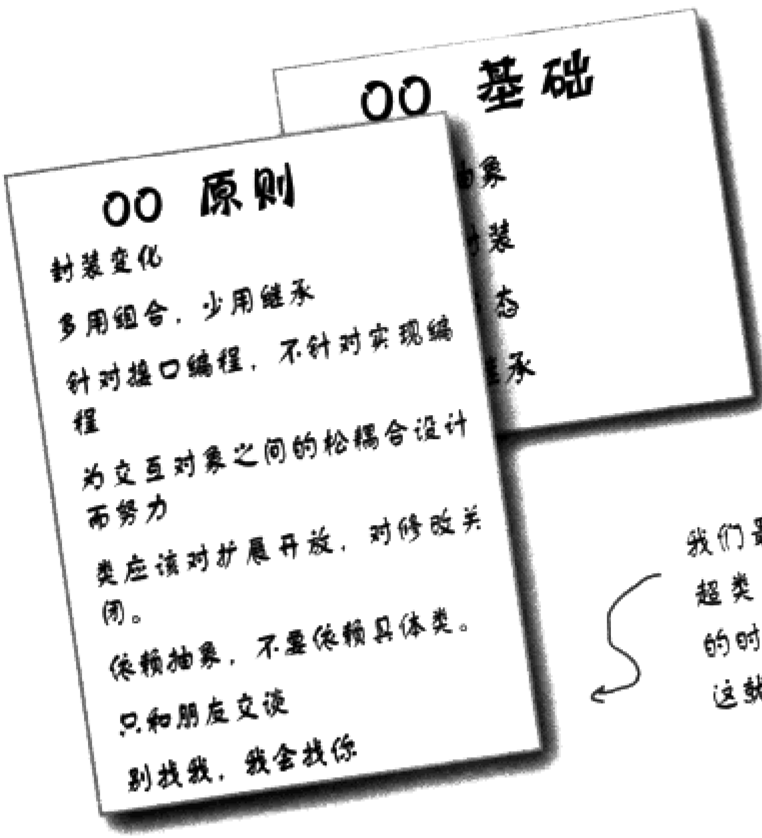
竖排提示：

2. __________ algorithm steps are implemented by hook methods
3. Factory Method is a ________________ of Template Method
7. The steps in the algorithm that must be supplied by the subclasses are usually declared ____________
8. Huey, Louie and Dewey all weigh _________ pounds
9. A method in the abstract superclass that does nothing or provides default behavior is called a __________ method
10. Big headed pattern
11. Our favorite coffee shop in Objectville
15. The Arrays class implements its template method as a __________ method



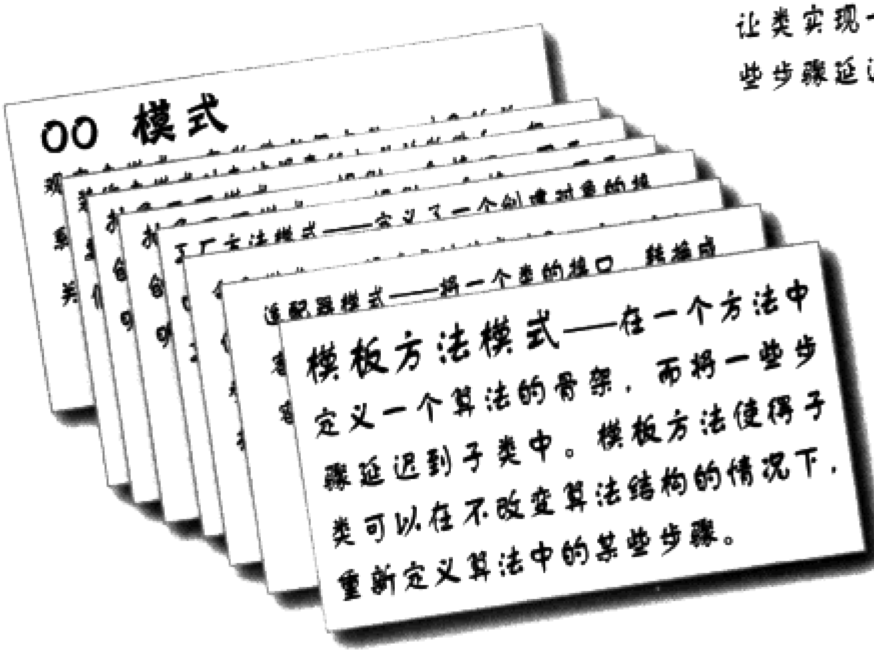
设计箱内的工具

我们在你的工具箱内放进模板方法模式。有了模板方法，你就可以像专家一样复用代码，同时保持对算法的控制。



我们最新的原则提醒你，由超类主控一切，当它们需要的时候，自然会去调用子类，这就跟好莱坞一样。

还有我们最新的模式，让类实现一个算法，将一些步骤延迟到子类。



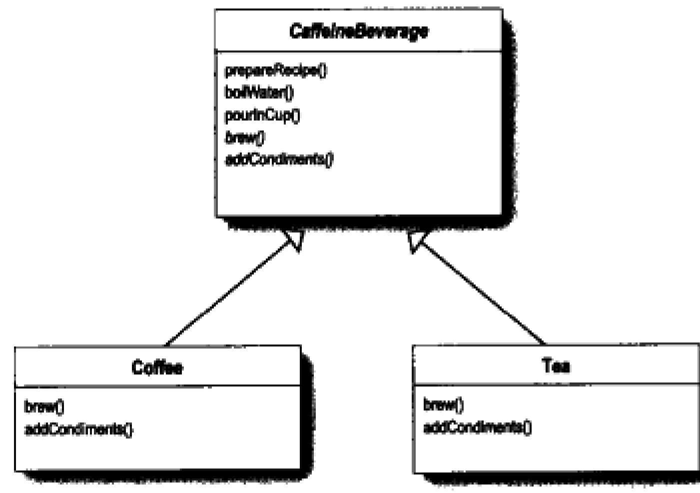
要点

- “模板方法”定义了算法的步骤，把这些步骤的实现延迟到子类。
- 模板方法模式为我们提供了一种代码复用的重要技巧。
- 模板方法的抽象类可以定义具体方法、抽象方法和钩子。
- 抽象方法由子类实现。
- 钩子是一种方法，它在抽象类中不做事，或者只做默认的事情，子类可以选择要不要去覆盖它。
- 为了防止子类改变模板方法中的算法，可以将模板方法声明为final。
- 好莱坞原则告诉我们，将决策权放在高层模块中，以便决定如何以及何时调用低层模块。
- 你将在真实世界代码中看到模板方法模式的许多变体，不要期待它们全都是一眼就可以被你认出的。
- 策略模式和模板方法模式都封装算法，一个用组合，一个用继承。
- 工厂方法是模板方法的一种特殊版本。

习题解答

Sharpen your pencil

绘制这个新的类图。我们已经将prepareRecipe()的实现移到CaffeineBeverage类中：

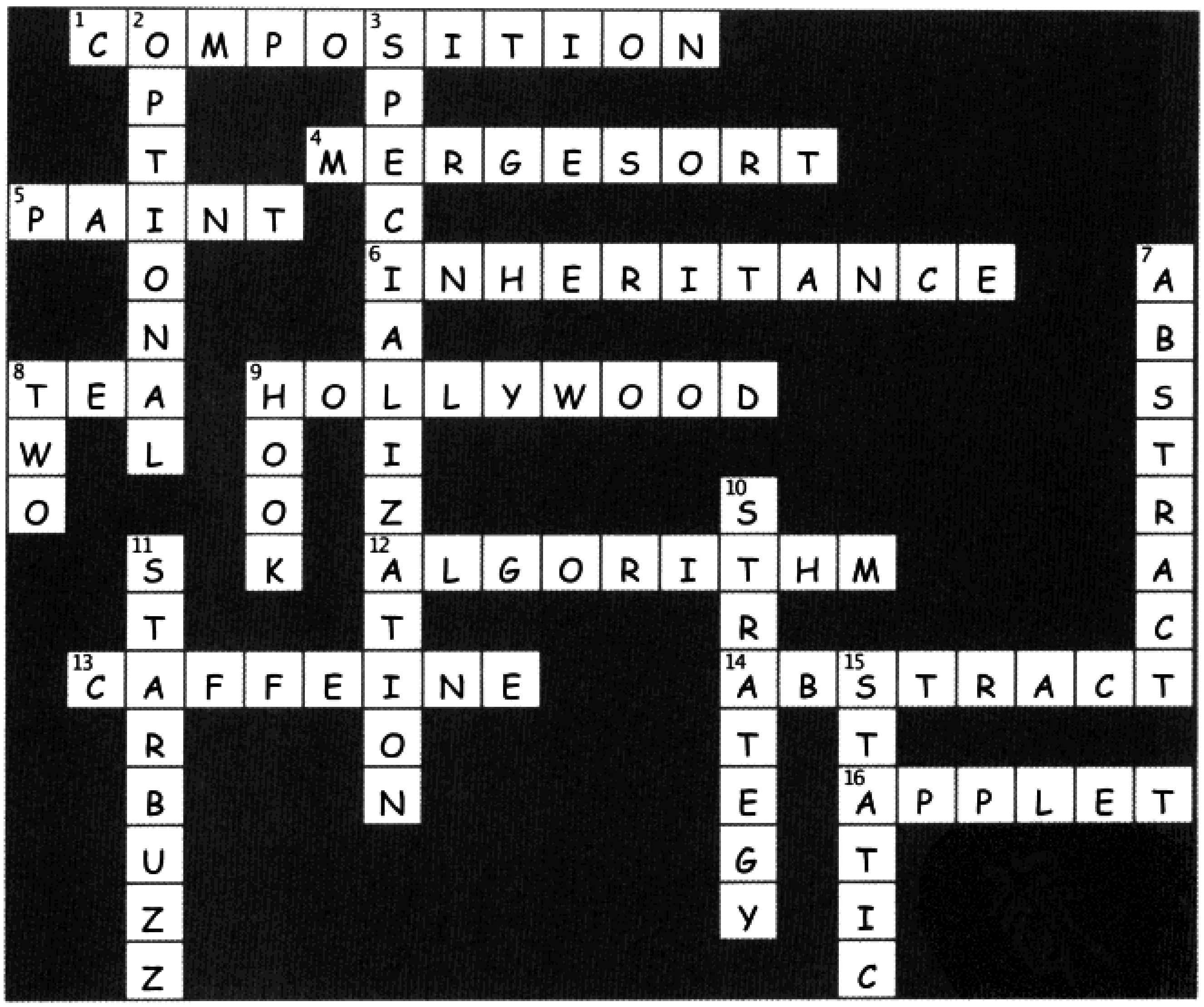


连连看

将模式和叙述之间配对：

模式	叙述
模板方法	封装可互换的行为，然后使用委托来决定要采用哪一个行为
策略	子类决定如何实现算法中的某些步骤
工厂方法	由子类决定实例化哪个具体类

习题解答


1 COMPOSITION
2 OPTIONAL
3 SPECIALIZATION
4 MERGESORT
5 PAINT
6 INHERITANCE
7 ABSTRACT
8 TEA
8 TWO
9 HOLLYWOOD
9 HOOK
10 STRATEGY
11 STARBUZZ
12 ALGORITHM
13 CAFFEINE
14 ABSTRACT
15 STATIC
16 APPLET

9 迭代器与组合模式

管理良好的集合



有许多种方法可以把对象堆起来成为一个集合（collection）。你可以把它们放进数组、堆栈、列表或者是散列表（Hashtable）中，这是你的自由。每一种都有它自己的优点和适合的使用时机，但总有一个时候，你的客户想要遍历这些对象，而当他这么做时，你打算让客户看到你的实现吗？我们当然希望最好不要！这太不专业了。没关系，不要为你的工作担心，你将在本章中学习如何能让客户遍历你的对象而又无法窥视你存储对象的方式；也将学习如何创建一些对象超集合（super collection），能够一口气就跳过某些让人望而生畏的数据结构。你还将学到一些关于对象职责的知识。

爆炸性新闻：对象村餐厅和对象村煎饼屋合并了

真是个好消息！现在我们可以在同一个地方，享用煎饼屋美味的煎饼早餐，和好吃的餐厅午餐了。但是，好像有一点小麻烦……



检查菜单项

至少Lou和Mel都同意实现MenuItem。
让我们检查每份菜单上的项目和实现。



餐厅的菜单有许多午餐项目，而煎饼屋的菜单则是早餐项目。每个菜单项都有名称、叙述和价格。

```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

菜单项包含了名称、叙述、是否为素食的标志，还有价格。将这些值传入构造器来初始化这个菜单项。

这些getter方法让你能够取得菜单项的各个字段。

Lou和Mel的菜单实现

让我们看看Lou和Mel在吵些什么。他们都在菜单项的存储方式上花了很多的时间和代码，也许有许多其他代码依赖这些菜单项。

我用的是ArrayList，这样才可以轻易地扩展菜单。



这是Lou的煎饼屋菜单实现。

```
public class PancakeHouseMenu {
    ArrayList menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }
    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList getMenuItems() {
        return menuItems;
    }

    // 这里还有菜单的其他方法
}
```

Lou使用一个ArrayList存储他的菜单项。

在菜单的构造器中，每一个菜单项都会被加入到ArrayList中。

每一个菜单项都有一个名称、一个叙述、是否为素食项，还有价格。

要加入一个菜单项，Lou的做法是：创建一个新的菜单项对象，传入每一个变量，然后将它加入此ArrayList中。

这个getMenuItems()方法返回菜单项列表。

Lou还有许多其他的菜单代码，它们都依赖于这个ArrayList，所以他不希望重写全部的代码！



Mel的餐厅菜单是这么实现的。

```
public class DinerMenu{
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
            "A hot dog, with saurkraut, relish, onions, topped with cheese",
            false, 3.05);
        // 继续加入其他项目
    }

    public void addItem(String name, String description,
                         boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    // 这里还有菜单的其他方法
}
```

Mel采用不同的方法。他使用的是一个数组，所以可以控制菜单的长度，并且在取出菜单项的时候，不需要转型。

就跟Lou一样，Mel是使用addItem()辅助方法在构造器中创建菜单项的。

addItem()方法需要拿所有必要的参数来创建一个菜单项，并实例化它。这个方法也会检查数组是否已经超出了它的长度限制。

Mel特别坚持让他的菜单保持在一定的长度之内（或许是他不希望记太多食谱）。

getMenuItem()返回一个菜单项的数组。

就和Lou一样，Mel还有许多其他的菜单代码依赖于这个数组。他忙着煮菜，没空重写这么多代码！

有两种不同的菜单表现方式，这会带来什么问题？

想了解为什么有两种不同的菜单表现方式会让事情变得复杂化，让我们试着实现一个同时使用这两个菜单的客户代码。假设你已经被他们两个人合组的新公司雇用，你的工作是要创建一个Java版本的女招待（毕竟，这是对象村）。这个Java版本的女招待规格是:能应对顾客的需要打印定制的菜单，甚至告诉你是否某个菜单项是素食的，而无需询问厨师。这可是一大创新！

跟我们来看看这份关于女招待的规格，然后看看如何实现她……

女招待是Java版的



Java版本的女招待规格

Java版本的女招待：代号为“Alice”

printMenu()
- 打印出菜单上的每一项

printBreakfastMenu()
- 只打印早餐项

printLunchMenu()
- 只打印午餐项

printVegetarianMenu()
- 打印所有的素食菜单项

isItemVegetarian(name)
- 指定项的名称，如果该项是素食的话，返回true，否则返回false

女招待的规格

我们先从实现printMenu()方法开始：

1. 打印每份菜单上的所有项，必须调用PancakeHouseMenu和DinerMenu的getMenuItem()方法，来取得它们各自的菜单项。请注意，两者的返回类型是不一样的。

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

方法看起来一样，但是调用所返回的结果却是不一样的类型。

早餐项是在一个ArrayList中，午餐项则是在一个数组中。

2. 现在，想要打印PancakeHouseMenu的项，我们用循环将早餐ArrayList内的项一一列出来。想要打印DinerMenu的项目，我们用循环将数组内的项一一列出来。

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

现在，我们必须实现两个不同的循环，个别处理这两个不同的菜单……

……处理ArrayList的循环……

……处理数组的循环。

3. 实现女招待中的其他方法，做法也都和这一页的方法相类似。我们总是需要处理两个菜单，并且用两个循环遍历这些项。如果还有第三家餐厅以不同的实现出现，我们就需要有三个循环。

Sharpen your pencil

根据我们的printMenu()实现，下列哪一项为真？

- ❑ A. 我们是针对PancakeHouseMenu和DinerMenu的具体实现编码，而不是针对接口。
- ❑ B. 女招待没有实现Java女招待API，所以她没有遵守标准。
- ❑ C. 如果我们决定从DinerMenu切换到另一种菜单，此菜单的项是用Hashtable来存放的，我们会因此需要修改女招待中的许多代码。
- ❑ D. 女招待需要知道每个菜单如何表达内部的菜单项集合，这违反了封装。
- ❑ E. 我们有重复的代码：printMenu()方法需要两个循环，来遍历两种不同的菜单。如果我们加上第三种菜单，我们就需要第三个循环。
- ❑ F. 这个实现并没有基于MXML（Menu XML），所以就没有办法互操作。

下一步呢？

Mel和Lou让我们很为难。他们都不想改变自身的实现，因为意味着要重写许多代码。但是如果他们其中一人不肯退让，我们就很难办了，我们所写出来的女招待程序将难以维护、难以扩展。

如果我们能够找出一个方法，让他们的菜单实现一个相同的接口，该有多好！（除了他们的getMenuItem()方法的返回类型不同之外，这两个菜单其实非常类似）。这样一来，我们就可以最小化女招待代码中的具体引用，同时还有希望摆脱遍历这两个菜单所需的多个循环。

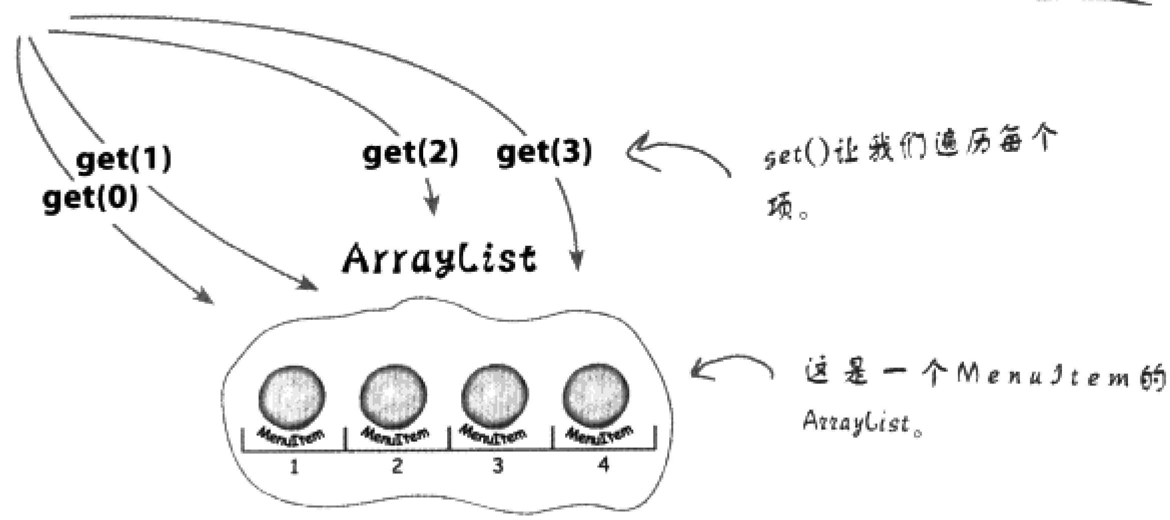
听起来很棒！但要怎么做呢？

可以封装遍历吗？

如果你从本书中学到了一件事情，那就是封装变化的部分。很明显，在这里发生变化的是：由不同的集合（collection）类型所造成的遍历。但是，这能够被封装吗？让我们来看看这个想法……

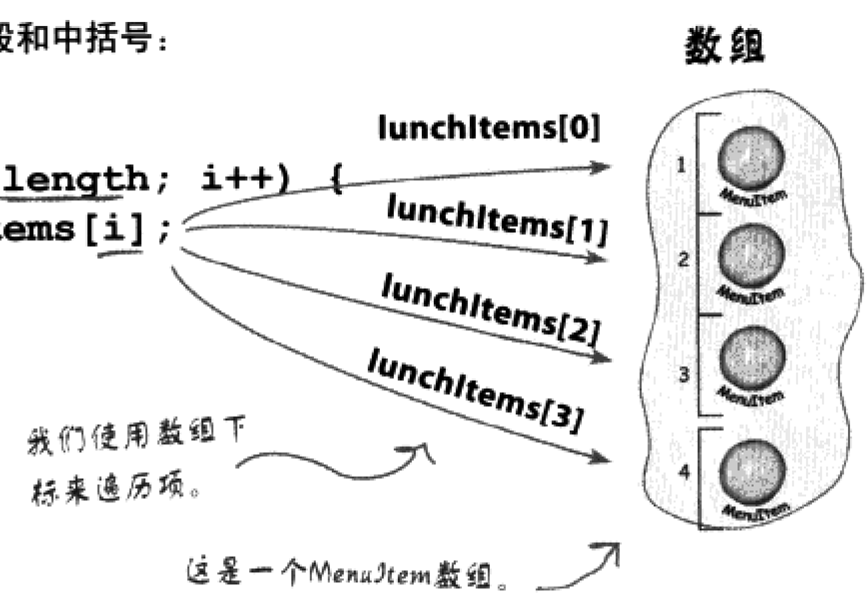
❶ 要遍历早餐项，我们需要使用ArrayList的size()和get()方法：

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



❷ 要遍历午餐项，我们需要使用数组的length字段和中括号：

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



3 现在我们创建一个对象，将它称为迭代器（Iterator），利用它来封装“遍历集合内的每个对象的过程”。先让我们在ArrayList上试试：

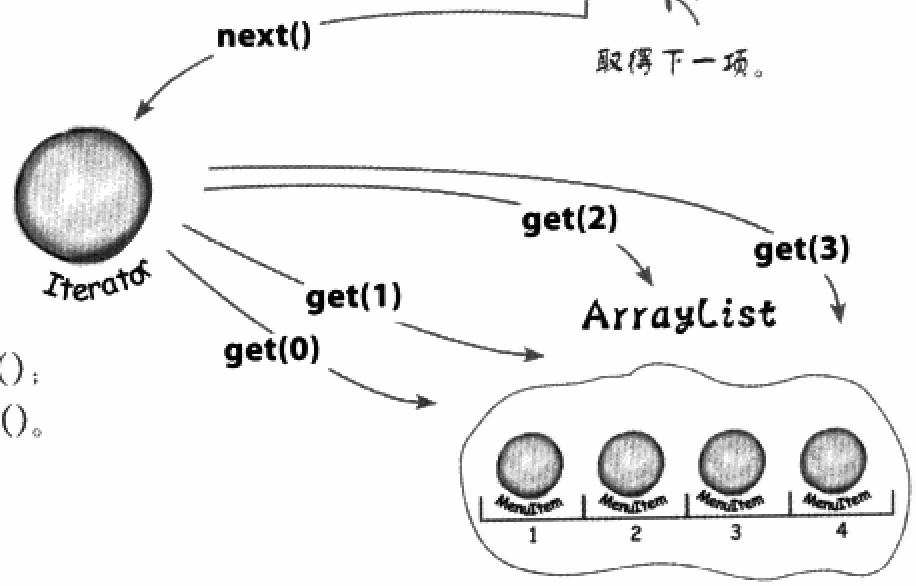
我们从breakfastMenu中取得一个菜单项迭代器。

```
Iterator iterator = breakfastMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

当还有其他项时……

取得下一项。



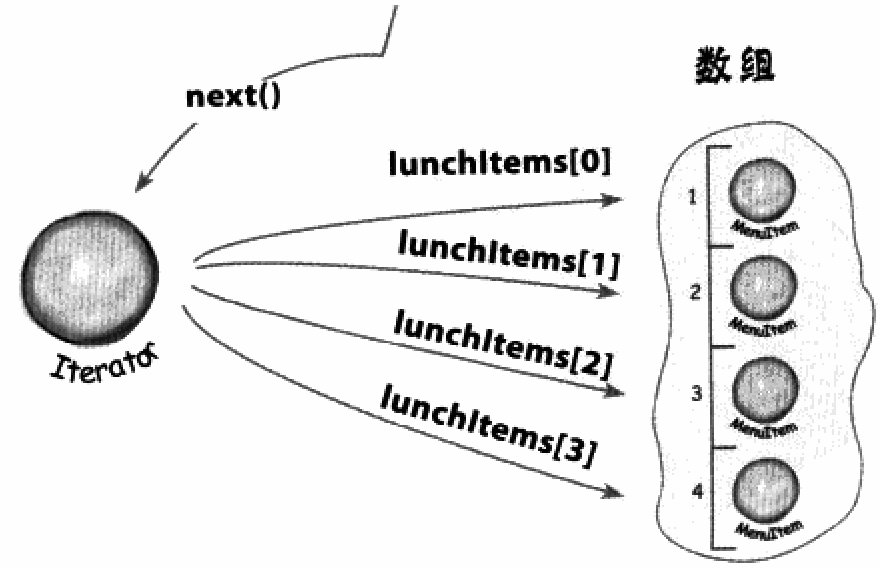
客户只需要调用hasNext()和next()；而迭代器会暗中调用ArrayList的get()。

4 将它也在数组上试试：

```
Iterator iterator = lunchMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

哇塞！这个代码和上面的breakfastMenu代码完全一样。

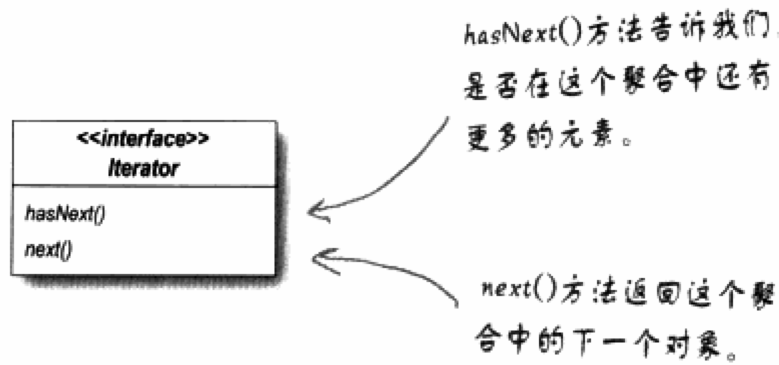


这里的情况也是一样的：客户只需调用hasNext()和next()即可；而迭代器会暗中使用数组的下标。

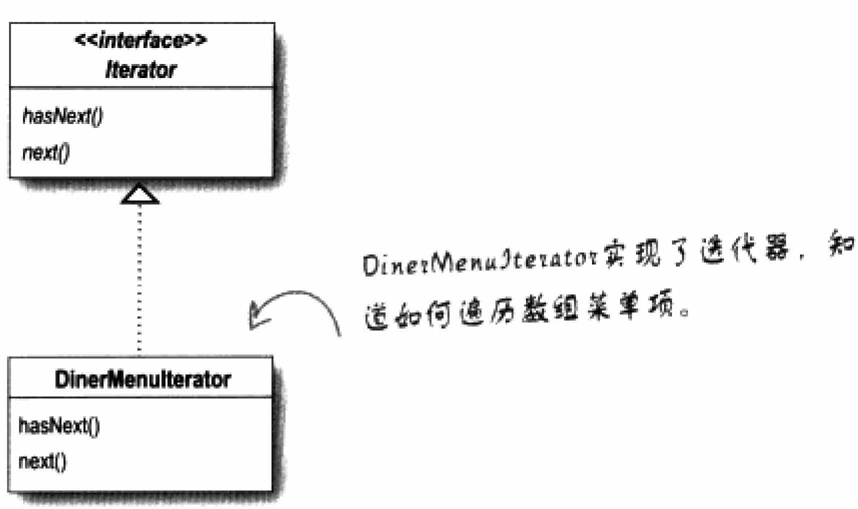
会见迭代器模式

看起来我们对遍历的封装已经奏效了；你大概也已经猜到，这正是一个设计模式，称为迭代器模式（Iterator Pattern）。

关于迭代器模式，你所需要知道的第一件事情，就是它依赖于一个名为迭代器的接口。这是一个可能的迭代器的接口：



现在，一旦我们有了这个接口，就可以为各种对象集合实现迭代器：数组、列表、散列表……如果我们想要为数组实现迭代器，以便使用在DinerMenu中，看起来就像这样：





让我们继续实现这个迭代器，并将它挂钩到DinerMenu中，看它是如何工作的……

在餐厅菜单中加入一个迭代器

想要在餐厅菜单中加入一个迭代器，我们需要先定义迭代器接口：

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

这是我们的两个方法：

其中，hasNext()方法会返回一个布尔值，让我们知道是否还有更多的元素……

……而next()方法返回下一个元素。

现在我们需要实现一个具体的迭代器，为餐厅菜单服务：

```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

实现迭代器接口。

position记录当前数组遍历的位置。

构造器需要被传入一个菜单项的数组当做参数。

next()方法返回数组内的下一项，并递增其位置。

hasNext()方法会检查我们是否已经取得数组内所有的元素。如果还有元素待遍历，则返回true。

因为使用的是固定长度的数组，所以我们不但要检查是否超出了数组长度，也必须检查是否下一项是null，如果是null，就表示没有其他项了。

用迭代器改写餐厅菜单

好了，我们已经有了迭代器。现在就利用它来改写餐厅菜单；我们只需加入一个方法创建一个DinerMenuIterator，并将它返回给客户：

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // 构造器在这里

    // addItem在这里

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }

    // 菜单的其他方法在这里
}
```

我们不再需要getMenuItems()方法，而且事实上，我们根本不想要这个方法，因为它会暴露我们内部的实现。

这是createIterator()方法，用来从菜单项数组创建一个DinerMenuIterator，并将它返回给客户。

返回迭代器接口。客户不需要知道餐厅菜单是如何维护菜单项的，也不需要知道迭代器是如何实现的。客户只需直接使用这个迭代器遍历菜单项即可。



习题

请继续完成PancakeHouseIterator的实现，并对PancakeHouseMenu类作出必要的修改。

修正女招待的代码

我们需要将迭代器代码整合进女招待中。我们应该摆脱原本冗余的部分。整合的做法相当直接：首先创建一个printMenu()方法，传入一个迭代器当做此方法的参数，然后对每一个菜单都使用createIterator()方法来检索迭代器，并将迭代器传入新方法。



经迭代器改良成新版的我。

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // 其他的方法
}
```

在构造器中，女招待照顾两个菜单。

这个printMenu()方法为每一个菜单各自创建一个迭代器。

然后对每个迭代器调用重载的(overloaded) printMenu()，将迭代器传入。

测试是否还有其他项。

取得下一项。

这个重载的printMenu()方法，使用迭代器，来遍历菜单项并打印出来。

使用该项来取得名称、价格和叙述，并打印出来。

现在我们只需要一个循环就可以了。

测试我们的代码

快来测试吧！让我们写一些测试程序，然后看看女招待如何工作……

首先我们创建了新的菜单。

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

        waitress.printMenu();
    }
}
```

然后我们创建了一个女招待，并将菜单传送给她。

然后我们把菜单打印出来。

执行结果……

```
File Edit Window Help GreenEggs&Ham
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

%
```

首先我们遍历煎饼屋菜单

然后是午餐菜单，所用的遍历代码相同

到目前为止，我们做了些什么？



首先，我们让对象村的厨师们非常快乐。他们可以保持他们自己的实现又可以摆平差别。只要我们给他们这两个迭代器（PancakeHouseMenuIterator和DinerMenuIterator），他们只需要加入一个createIterator()方法，一切就大功告成了。

这个过程中，我们也帮了我们自己。女招待将会更容易维护和扩展。让我们来彻底检查一下到底我们做了哪些事，以及后果如何：

难以维护的女招待实现	由迭代器支持的新女招待
菜单封装得不好；餐厅使用的是ArrayList，而煎饼屋使用的是数组。	菜单的实现已经被封装起来了。女招待不知道菜单是如何存储菜单项集合的。
需要两个循环来遍历菜单项。	只要实现迭代器，我们只需要一个循环，就可以多态地处理任何项的集合。
女招待捆绑于具体类（MenuItem[]和ArrayList）。	女招待现在只使用一个接口（迭代器）。
女招待捆绑于两个不同的具体菜单类，尽管这两个类的接口大致上是一样的。	现在菜单的接口完全一样，但是，哎呀！我们还是没有一个共同的接口，也就是说女招待仍然捆绑于两个具体的菜单类。这一点我们最好再修改一下。

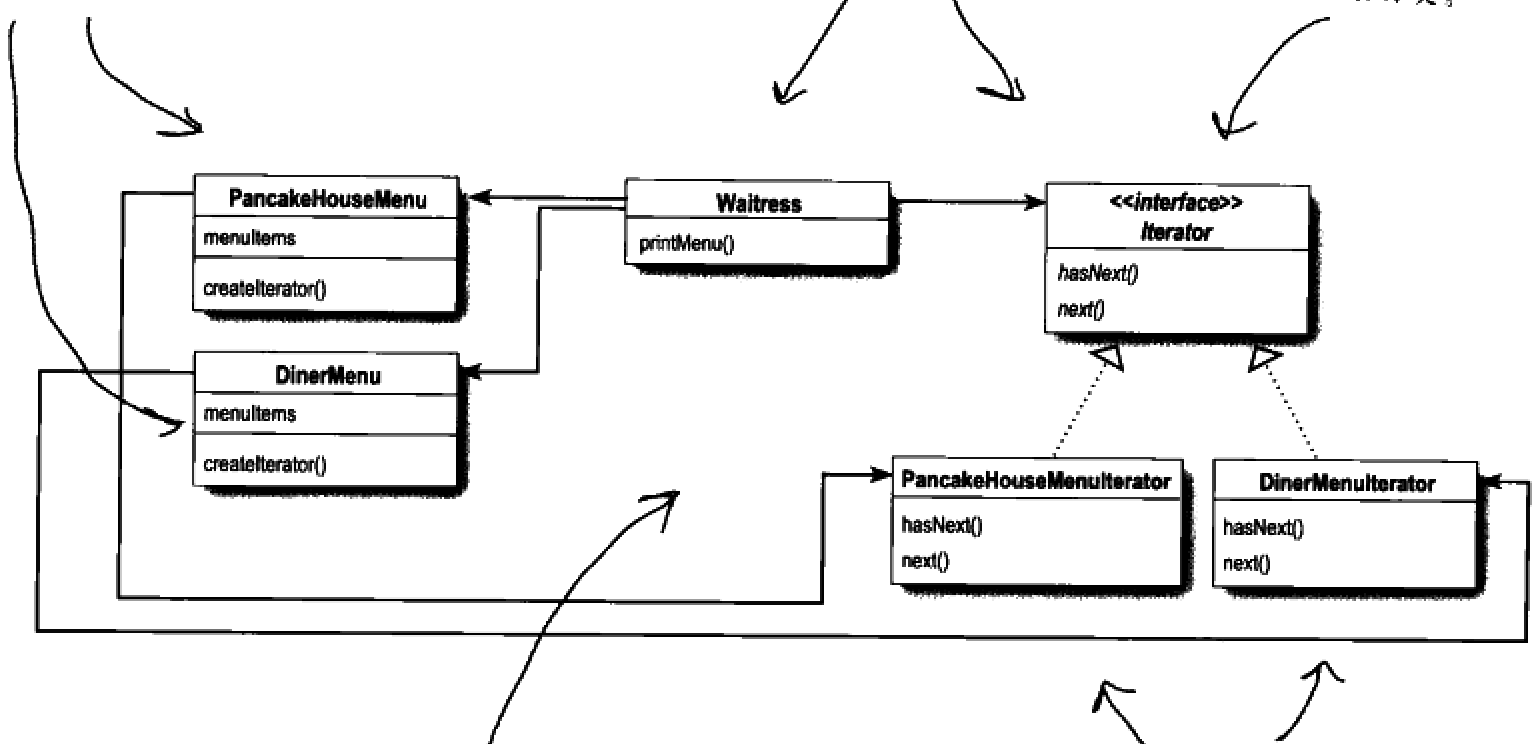
到目前为止，我们有些什么……

在清理一切之前，让我们从整体上来看看目前的设计。

这两个菜单都实现一样的方法，但是并没有实现相同的接口。我们将修改这一点，好让女招待不会依赖于具体的菜单。

这个迭代器让女招待能够从具体类的实现中解耦。她不需要知道菜单是使用数组、ArrayList，还是便利贴来实现。她只关心她能够取得迭代器。

我们现在使用一个共同的迭代器接口，实现了两个具体类。



请注意，迭代器让我们能够遍历聚合中的每个元素，而不会去强迫聚合必须提供方法，让我们在它的元素中游走。我们也可以在聚合的外面实现迭代器；换句话说，我们封装了遍历。

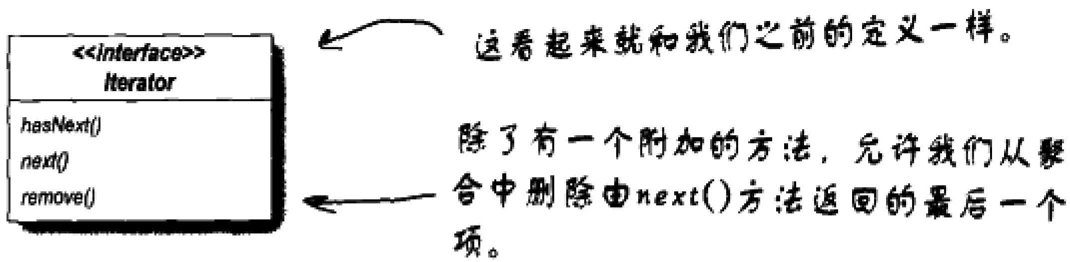
煎饼屋菜单和餐厅菜单都实现了新的createIterator()方法，它们负责为各自的菜单项实现创建迭代器。

做一些改良……

好了，我们已经知道这两份菜单的接口完全一样，但没有为它们设计一个共同的接口。所以，接下来就要这么做，让女招待更干净一些。

你可能会奇怪，为什么我们不使用Java的Iterator接口呢——我们之所以这么做，是为了要让你了解如何从头创建一个迭代器。现在我们的目的已经达到了，所以就要改变做法，开始使用Java的Iterator接口了。而这也会带来更多的好处。什么好处呢？很快你就会知道了。

首先，让我们看看java.util.Iterator接口：



这一切都太简单了：我们只需将煎饼屋菜单迭代器和餐厅菜单迭代器所扩展的接口，由我们自己的迭代器接口，改成java.util的迭代器接口即可，对吧？差不多就这样……实际上，甚至更简单。其实不只java.util有迭代器接口，连ArrayList也有一个返回一个迭代器的iterator()方法。换句话说，我们并不需要为ArrayList实现自己的迭代器。然而，我们仍然需要为餐厅菜单实现一个迭代器，因为餐厅菜单使用的是数组，而数组不支持iterator()方法（或其他创建数组迭代器的方法）。

---

there are no Dumb Questions

问：如果我不想让客户具备删除的能力，该怎么办？

答： remove()方法其实是可有可无的，不一定要提供删除的功能。但是，很明显的，你需要提供这样的方法，因为毕竟它被声明在Iterator接口中。如果你不允许remove()的话，可以抛出一个java.lang.UnsupportedOperationException运行时异常。

Iterator的API文件提到可以让remove()抛出这样的异常，而任何良好的客户程序只要调用了remove()方法，就应该检查是否会发生这个异常。

问：在多线程的情况下，可能会有多个迭代器引用同一个对象集合。remove()会造成怎样的影响？

答： 后果并没有指明，所以很难预料。当你的程序在多线程的代码中使用到迭代器时，必须特别小心。

这看起来就和我们之前的定义一样。

利用java.util.Iterator来清理

让我们先从煎饼屋菜单开始，先把它改用java.util.Iterator，这很容易，只需要删除煎饼屋菜单迭代器类，然后在煎饼屋菜单的代码前面加上import java.util.Iterator，再改变下面这一行代码就可以了：

```
public Iterator createIterator() {
    return menuItems.iterator();
}
```

不创建自己的迭代器，而是调用菜单项ArrayList的iterator()方法。

这样PancakeHouseMenu就完成了。

接着，我们处理DinerMenu，以符合java.util.Iterator的需求。

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator {
    MenuItem[] list;
    int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public Object next() {
        // 在这里实现
    }

    public boolean hasNext() {
        // 在这里实现
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

首先，导入java.util.Iterator，我们需要实现这个接口。

这部分都没有变动……

……但是我们需要实现remove()方法。因为使用的是固定长度的数组，所以在remove()被调用时，我们将后面的所有元素往前移动一个位置。

就快完成了……

我们只需要给菜单一个共同的接口，然后再稍微改一下女招待。这个Menu接口相当简单：可能迟早需要在里面多加入一些方法，例如addItem()，但是目前，我们还是让厨师控制他们的菜单，不要把那些方法放在公开接口中：

```
public interface Menu {
    public Iterator createIterator();
}
```

这是一个简单接口，让客户能够取得一个菜单项迭代器。

现在，我们需要让煎饼屋菜单类和餐厅菜单类都实现Menu接口，然后更新女招待的代码如下：

```
import java.util.Iterator;

public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // 其他的方法
}
```

现在女招待也使用java.util.Iterator。

将具体菜单类改成Menu接口。

这部分没有修改。

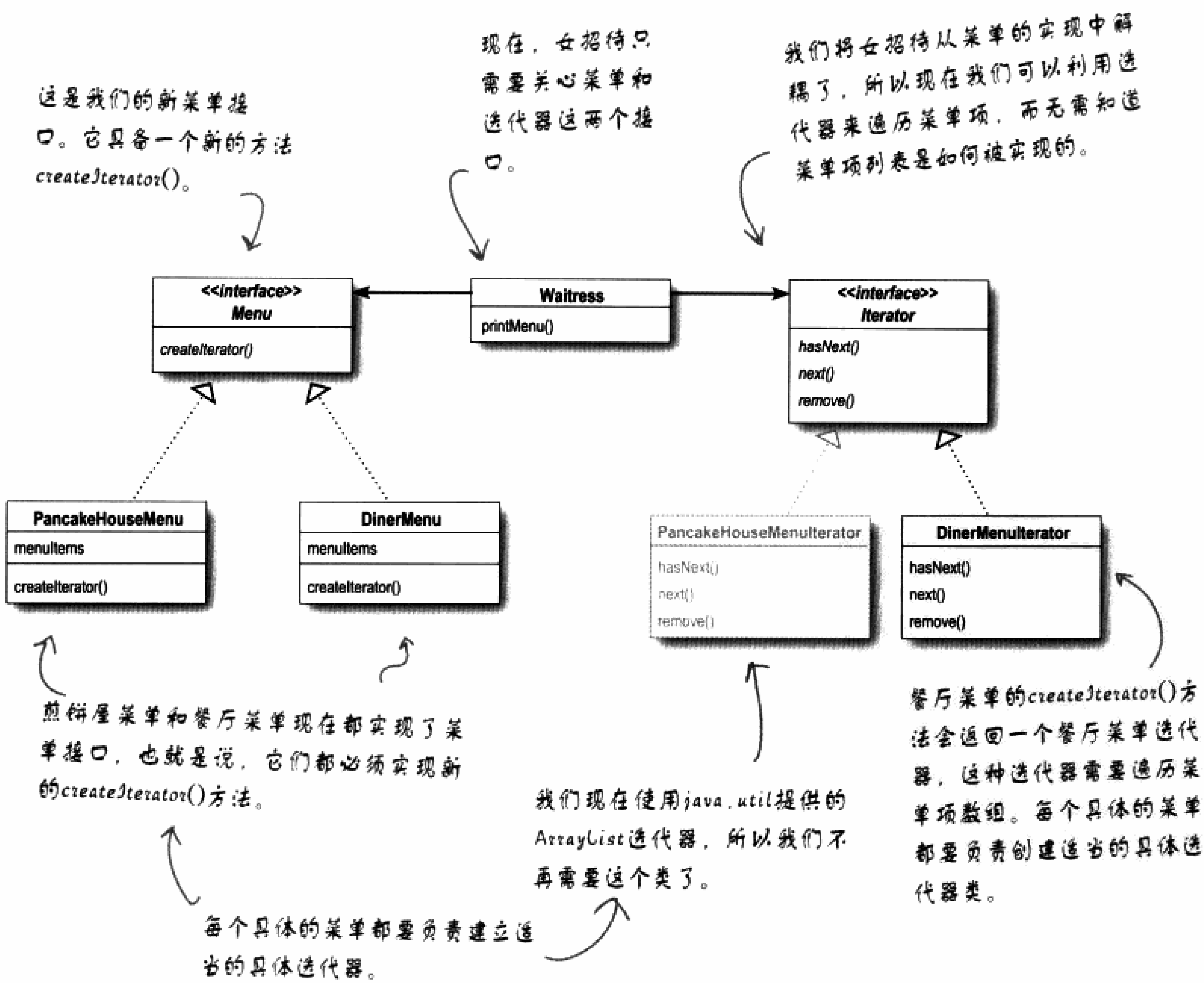
这为我们带来了什么好处？

煎饼屋菜单和餐厅菜单的类，都实现了Menu接口，女招待可以利用接口（而不是具体类）引用每一个菜单对象。这样，通过“针对接口编程，而不针对实现编程”，我们就可以减少女招待和具体类之间的依赖。

女招待依赖具体菜单的问题，这下子解决了！

这个新的菜单接口有一个方法，createIterator()。此方法是由煎饼屋菜单和餐厅菜单实现的。每个菜单类都必须负责提供适当的具体迭代器。

女招待依赖菜单项的实现的问题，这下子也解决了！



定义迭代器模式

你已经知道了如何用自己的迭代器来实现迭代器模式，也看到了Java是如何在某些面向聚合的类中（如ArrayList）支持迭代器的。现在我们就来看看这个模式的正式定义：

> **迭代器模式**提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

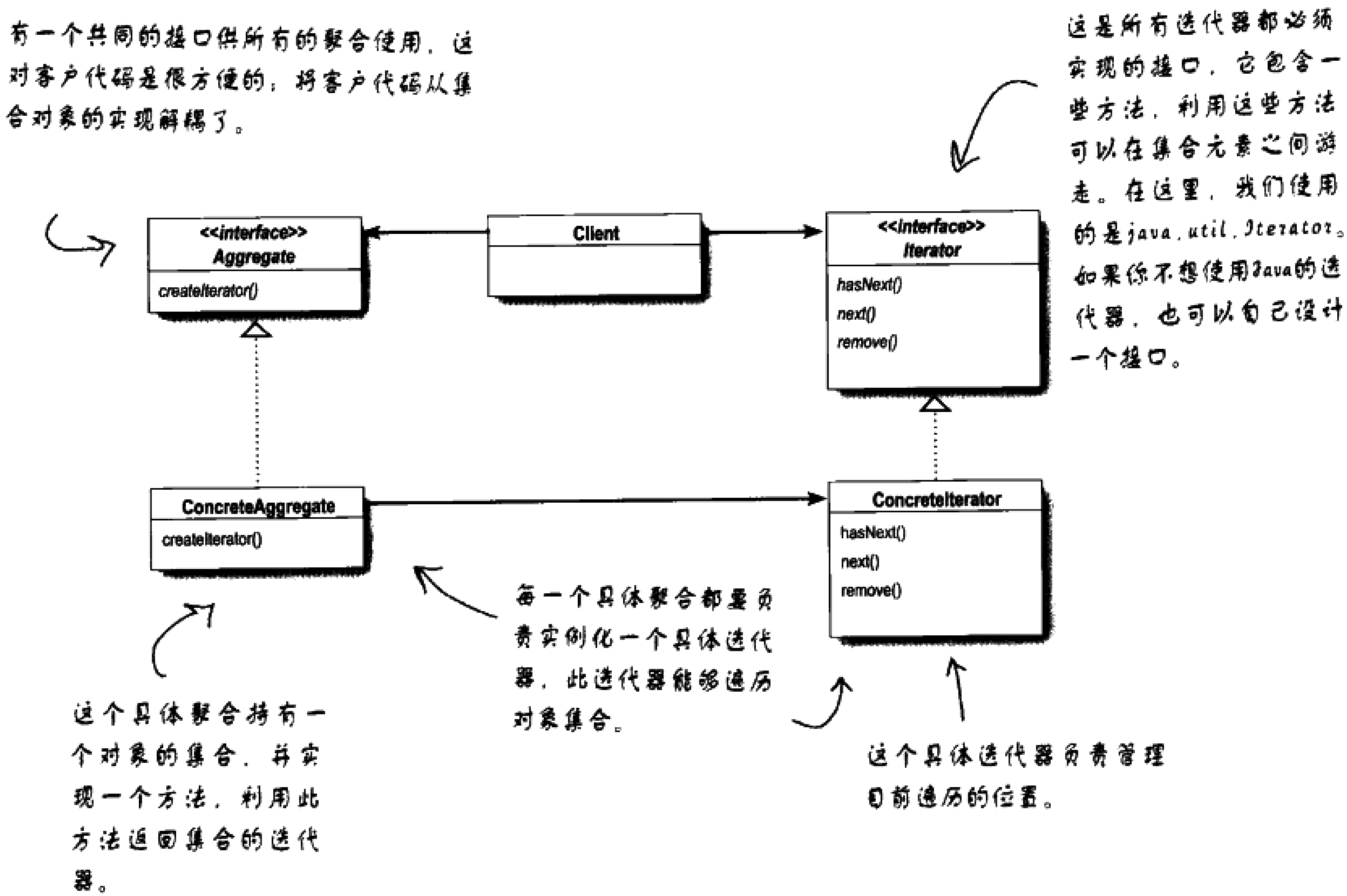
迭代器模式让我们能游走于聚合内的每一个元素，而又不暴露其内部的表示。

这很有意义：这个模式给你提供了一种方法,可以顺序访问一个聚集对象中的元素，而又不用知道内部是如何表示的。你已经在前面的两个菜单实现中看到了这一点。在设计中使用迭代器的影响是明显的：如果你有一个统一的方法访问聚合中的每一个对象，你就可以编写多态的代码和这些聚合搭配，使用——如同前面的printMenu()方法一样，只要有了迭代器这个方法根本不管菜单项究竟是由数组还是由ArrayList（或者其他能创建迭代器的东西）来保存的。

把游走的任务放在迭代器上，而不是聚合上。这样简化了聚合的接口和实现，也让责任各得其所。

另一个对你的设计造成重要影响的，是迭代器模式把在元素之间游走的责任交给迭代器，而不是聚合对象。这不仅让聚合的接口和实现变得更简洁，也可以让聚合更专注在它所应该专注的事情上面（也就是管理对象集合），而不必去理会遍历的事情。

让我们检查类图，将来龙去脉拼凑出来……



BRAIN POWER

迭代器模式的这张类图看起来很像我们所学过的另一个模式，你知道是哪个模式吗？提示：子类决定要创建哪个对象。

there are no Dumb Questions

问：我看到其他书上让迭代器类提供一些方法叫做first()、next()、isDone()和currentItem()。为什么这些方法不一样？

答： 这些是"经典的"方法名称，它们随着时间的流逝渐渐改变了，而现在我们在java.util.Iterator中所使用的名称有next()、hasNext()甚至remove()。

我们来看看这些经典的方法。java.util.Iterator将next()（移到下个位置）和currentItem()（取出目前的项目）合并成一个方法next()；isDone()变成了hasNext()；至于first()则不存在对应，这是因为在Java中，我们倾向于取得一个新的迭代器，而不是让目前的迭代器跳到一开始的位置。所以，其实这些接口没什么太大的差异。事实上，你还可以在迭代器内加入许多的方法，例如remove()方法。

问：我听说过"内部的"迭代器和"外部的"迭代器。这是什么？我们在前面例子中实现的是哪一种？

答：我们实现的是外部的迭代器，也就是说，客户通过调用next()取得下一个元素。而内部的迭代器则是由迭代器自己控制。在这种情况下，因为是由迭代器自行在元素之间游走，所以你必须告诉迭代器在游走的过程中，要做些什么事情，也就是说，你必须将操作传入给迭代器。因为客户无法控制遍历的过程，所以内部迭代器比外部迭代器更没有弹性。然而，某些人可能认为内部的迭代器比较容易使用，因为只需将操作告诉它，它就会帮你做完所有事情。

问：迭代器可以被实现成向后移动吗，就像向前移动一样？

答： 绝对可以。在这样的情况下，你可能要加上两个方法，一个方法取得前一个元素，而另一个方法告诉你是否已经到了集合的最前端。Java的Collection Framework 提供另一种迭代器接口，称为ListIterator。这个迭代器在标准的迭代器接口上多加了一个previous() 和一些其他的方法。任何实现了List接口的集合，都支持这样的做法。

问：对于散列表这样的集合，元素之间并没有明显的次序关系，我们该怎么办？

答： 迭代器意味着没有次序。只是取出所有的元素，并不表示取出元素的先后就代表元素的大小次序。对于迭代器来说，数据结构可以是有次序的，或是没有次序的，甚至数据可以是重复的。除非某个集合的文件有特别说明，否则不可以对迭代器所取出的元素大小顺序作出假设。

问：你说可以用迭代器写出"多态的代码"，可以再多做一些解释吗？

答： 当我们写了一个需要以迭代器当做参数的方法时，其实就是在使用多态的迭代。也就是说，我们所写出的代码，可以在不同的集合中游走，只要这个集合支持迭代器即可。我们不在乎这个集合是如何被实现的，但依然可以编程在它内部的元素之间游走。

问：如果我使用Java，我不见得总是想要利用java.util.Iterator，可能想要使用自己的迭代器实现，和这些已经使用Java标准的迭代器的类做整合，这做得到吗？

答： 或许可以吧。如果你有一个通用的迭代器接口，那么让你自己的集合和Java的集合（例如ArrayList、Vector）混合使用就会比较容易。但是请记住，如果你需要在迭代器接口为你的集合新增功能，你可以随时扩展迭代器接口。

问：我看到Java有一个Enumeration（枚举）接口，它实现了迭代器模式吗？

答： 我们曾经在适配器的那一章中提到过这个接口，还记得吗？java.util.Enumeration是一个有次序的迭代器实现，它有两个方法，hasMoreElements()类似hasNext()，而nextElement()类似next()。然而，你应该比较想使用迭代器，而不是枚举，因为大多数的Java类都支持迭代器。如果你想把这两者互相转换，请复习适配器那一章，在那一章里你实现了枚举和迭代器的适配器。

单一责任

如果我们允许我们的聚合实现它们内部的集合，以及相关的操作和遍历的方法，又会如何？我们已经知道这会增加聚合中的方法个数，但又怎样呢？为什么这么做不好？

想知道为什么，首先你需要认清楚，当我们允许一个类不但要完成自己的事情（管理某种聚合），还同时要担负更多的责任（例如遍历）时，我们就给了这个类两个变化的原因。两个？没错，就是两个：如果这个集合改变的话，这个类也必须改变；如果我们遍历的方式改变的话，这个类也必须跟着改变。所以，再一次地，我们的老朋友“改变”又成了我们设计原则的中心：

> **设计原则**
>
> 一个类应该只有一个引起变化的原因

我们知道要避免类内的改变，因为修改代码很容易造成许多潜在的错误。如果有一个类具有两个改变的原因，那么这会使得将来该类的变化机率上升，而当它真的改变时，你的设计中同时有两个方面将会受到影响。

要如何解决呢？这个原则告诉我们将一个责任只指派给一个类。

没错，这听起来很容易，但其实做起来并不简单：区分设计中的责任，是最困难的事情之一。我们的大脑很习惯看着一大群的行为，然后将它们集中在一起，尽管他们可能属于两个或多个不同的责任。想要成功的唯一方法，就是努力不懈地检查你的设计，随着系统的成长，随时观察有没有迹象显示某个类改变的原因超出一个。

类的每个责任都有改变的潜在区域。超过一个责任，意味着超过一个改变的区域。

这个原则告诉我们，尽量让每个类保持单一责任。



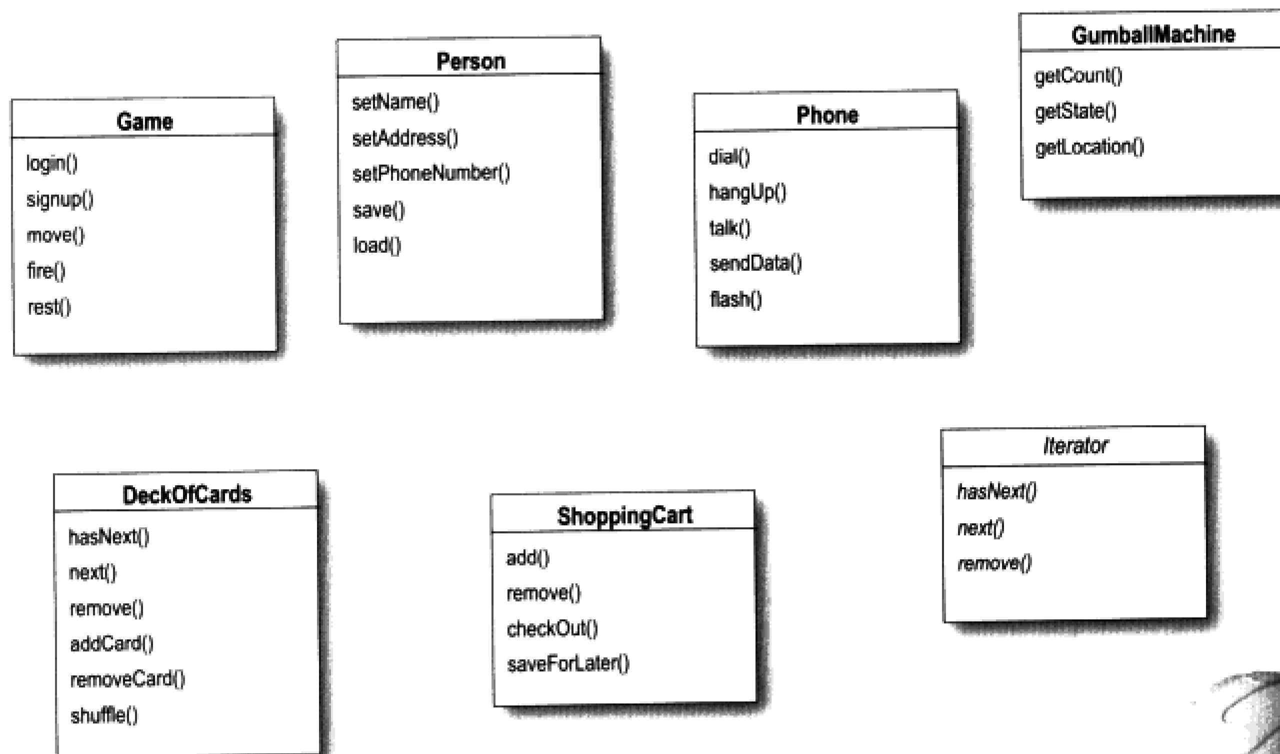
内聚（cohesion）这个术语你应该听过，它用来度量一个类或模块紧密地达到单一目的或责任。

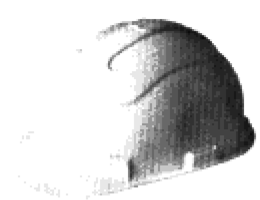
当一个模块或一个类被设计成只支持一组相关的功能时，我们说它具有高内聚；反之，当被设计成支持一组不相关的功能时，我们说它具有低内聚。

内聚是一个比单一责任原则更普遍的概念，但两者其实关系是很密切的。遵守这个原则的类容易具有很高的凝聚力，而且比背负许多责任的低内聚类更容易维护。

BRAIN POWER

研究这些类，并找出其中哪些类具有多重责任。

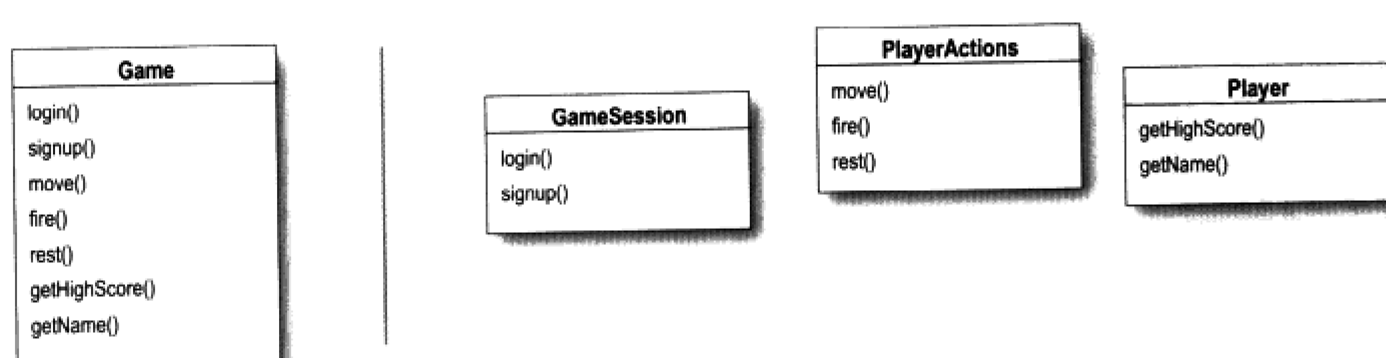


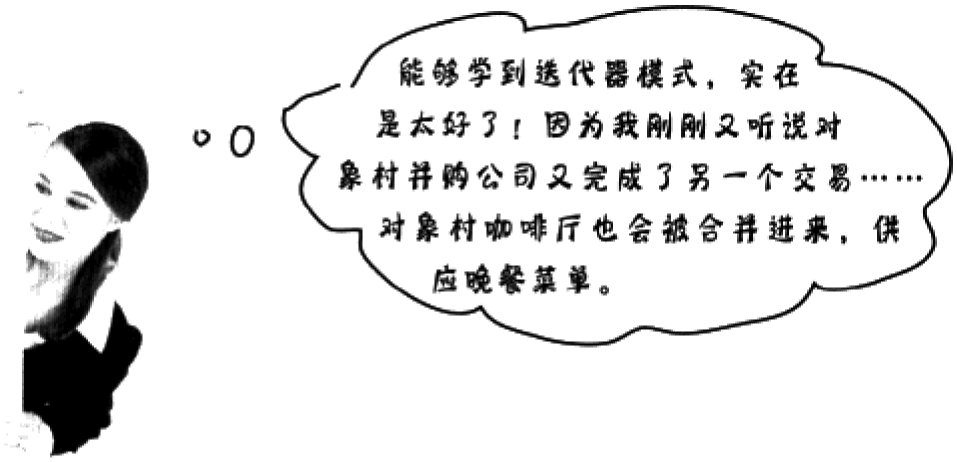


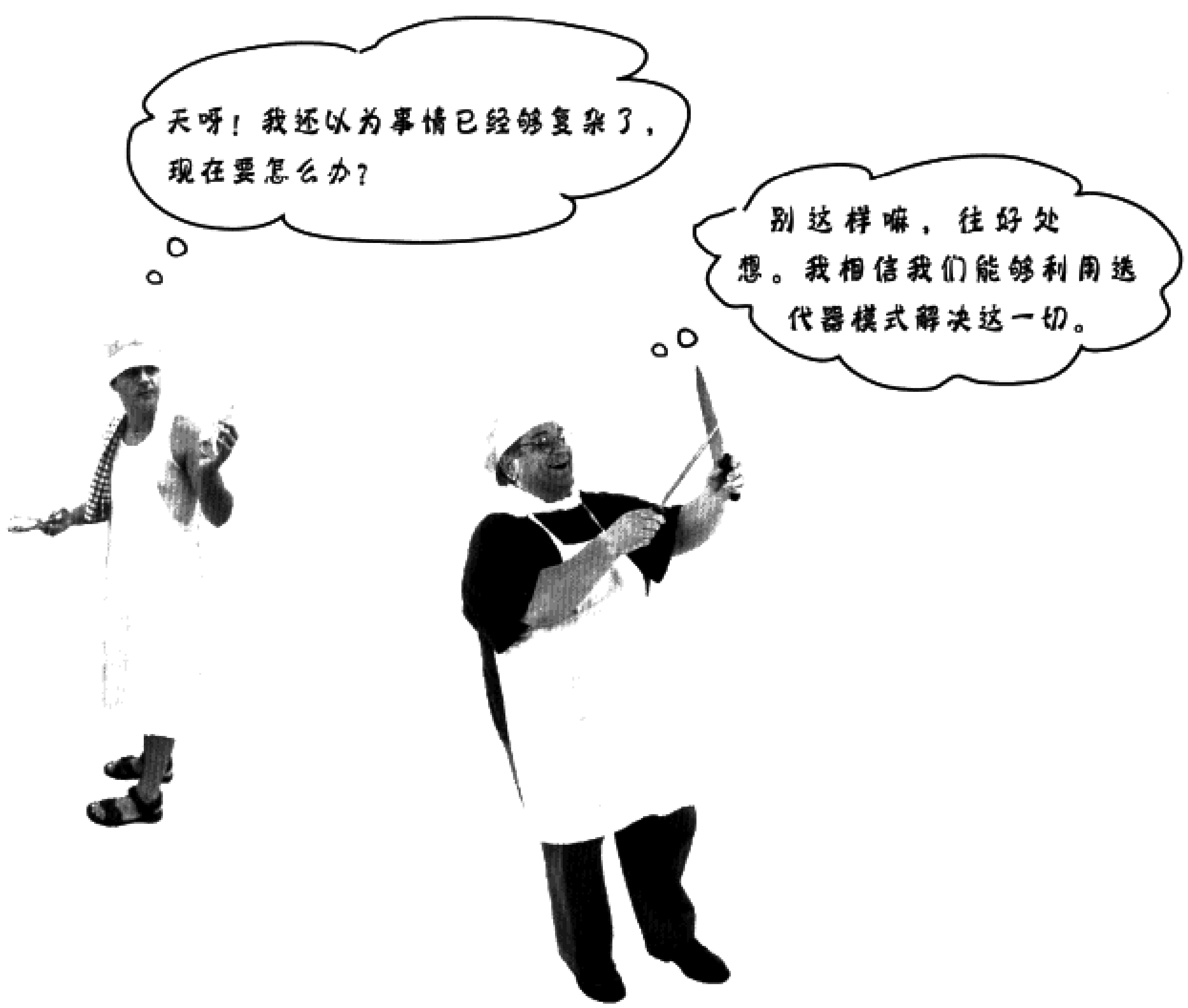
施工区!!
注意落物伤人

BRAIN 2 POWER

决定这些类的内聚是高或低。




能够学到迭代器模式，实在是太好了！因为我刚刚又听说对象村并购公司又完成了另一个交易……对象村咖啡厅也会被合并进来，供应晚餐菜单。


天呀！我还以为事情已经够复杂了，现在要怎么办？
别这样嘛，往好处想。我相信我们能够利用迭代器模式解决这一切。

看看咖啡厅的菜单

这是咖啡厅的菜单，要把这个菜单整合进我们的框架中，似乎不是太难的事情……看看怎么做。

咖啡厅菜单并没有实现我们的Menu接口，但这很容易修改。

菜单项是用散列表存储的，不知道这是否支持迭代器？等一下研究看看……

就跟其他的菜单一样，菜单项在构造器中初始化。

```
public class CafeMenu {
    Hashtable menuItems = new Hashtable();

    public CafeMenu() {
        addItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99);
        addItem("Soup of the day",
            "A cup of the soup of the day, with a side salad",
            false, 3.69);
        addItem("Burrito",
            "A large burrito, with whole pinto beans, salsa, guacamole",
            true, 4.29);
    }

    public void addItem(String name, String description,
                         boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Hashtable getItems() {
        return menuItems;
    }
}
```

我们在这里创建新的菜单项，并将它加入到菜单项散列表中。

key是项目名称。

这个值就是菜单项对象

我们不再需要这个方法了。

Sharpen your pencil

在看下一页之前，请很快写下为了能让这份代码符合我们的框架，我们要对它做的三件事情：

1. ______________________________

2. ______________________________

3. ______________________________

重做咖啡厅代码

将咖啡厅菜单整合进我们的框架是很容易的。为什么呢？因为Hashtable本来就支持Java内置的迭代器。但是它和ArrayList有一些不同……

```
public class CafeMenu implements Menu {
    Hashtable menuItems = new Hashtable();

    public CafeMenu() {
        // 构造器的代码写在这里
    }

    public void addItem(String name, String description,
                         boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Hashtable getItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return menuItems.values().iterator();
    }
}
```

咖啡厅菜单实现Menu接口，所以女招待使用咖啡厅菜单的方式，就和其他的两个菜单没有两样。

我们使用Hashtable，因为这是一个很常见的存储值的数据结构。你也可以使用比较新的HashMap。

跟以前一样，我们可以避开getItems()，所以我们不需要对女招待暴露菜单项的实现。

我们在这里实现了createIterator()方法。注意，我们不是取得整个Hashtable的迭代器，而是取得值的部分的迭代器。

再靠近一点

Hashtable比起ArrayList复杂许多，因为它的每一笔数据都是由一个key和一个值所组成，尽管如此，我们还是可以获得值（也就是菜单项）的迭代器。

```
public Iterator createIterator() {
    return menuItems.values().iterator();
}
```

首先，我们取得Hashtable的值，也就是值所组成的集合。

很幸运，这个集合支持iterator()方法，该方法返回一个java.util.Iterator类型的对象。

让女招待认识咖啡厅菜单

如何修改女招待，让她能够支持我们的新菜单呢？现在女招待已经能够接受迭代器了，所以应该不困难。

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu cafeMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
        this.cafeMenu = cafeMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        Iterator cafeIterator = cafeMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
        System.out.println("\nDINNER");
        printMenu(cafeIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

这个咖啡厅菜单会和其他菜单一起被传入女招待的构造器中，然后记录在一个实例变量中。

我们使用这个咖啡厅的菜单作为晚餐的菜单。想要将菜单打印出来，我们只需将它传入printMenu()，就一切搞定！

这里不需要修改。

早餐、午餐和晚餐

让我们写一段程序来测试。

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu);

        waitress.printMenu();
}
```

创建一个咖啡厅菜单……

……然后将它传给女招待。

现在，我们打印时应该看到所有的三个菜单。

测试结果如下，看看新增的咖啡厅晚餐菜单部分！

```
File Edit Window Help Kathy&BertLikePancakes
% java DinerMenuTestDrive

MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

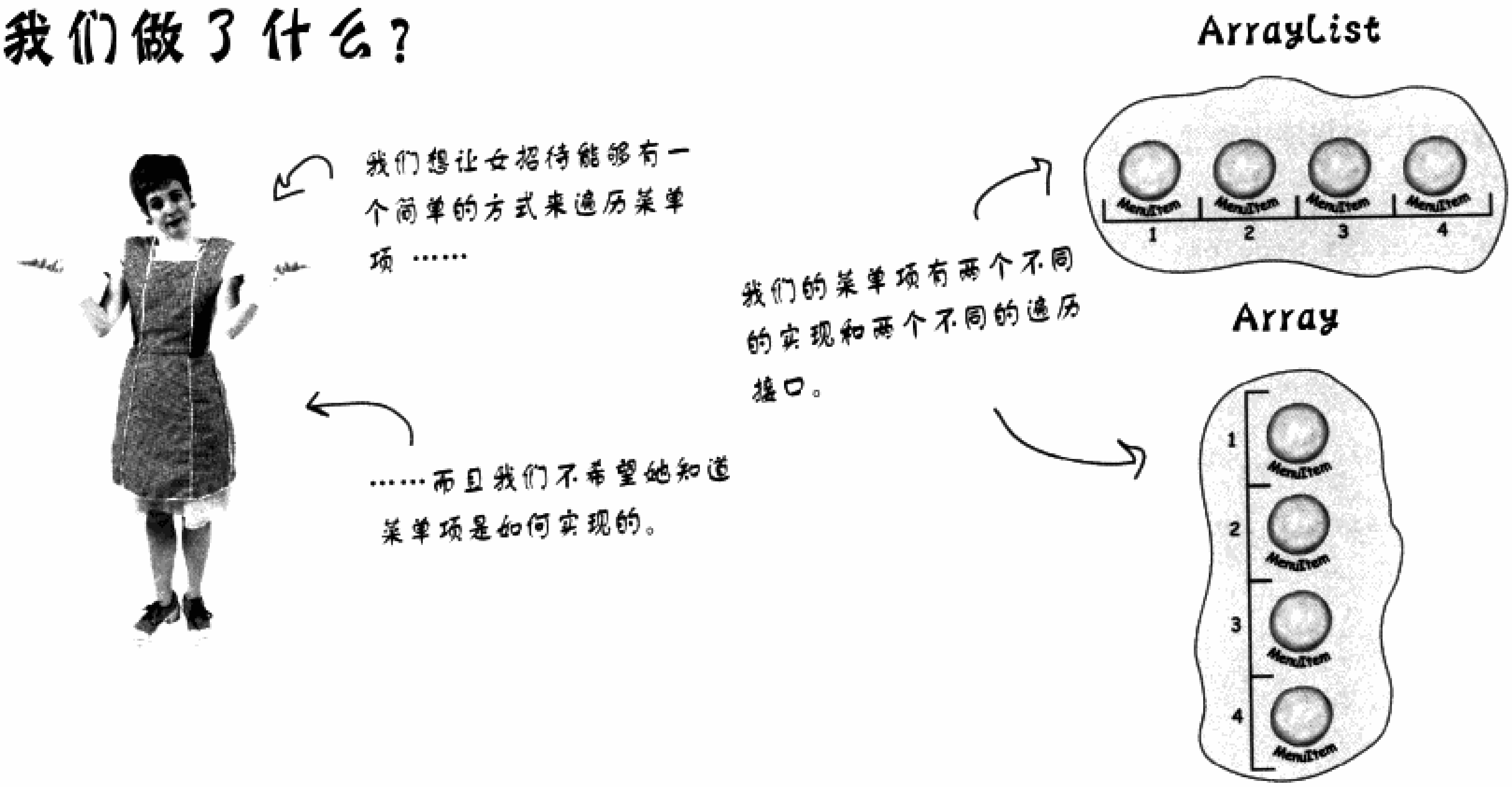
DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
 lettuce, tomato, and fries
%
```

首先我们遍历煎饼屋菜单。

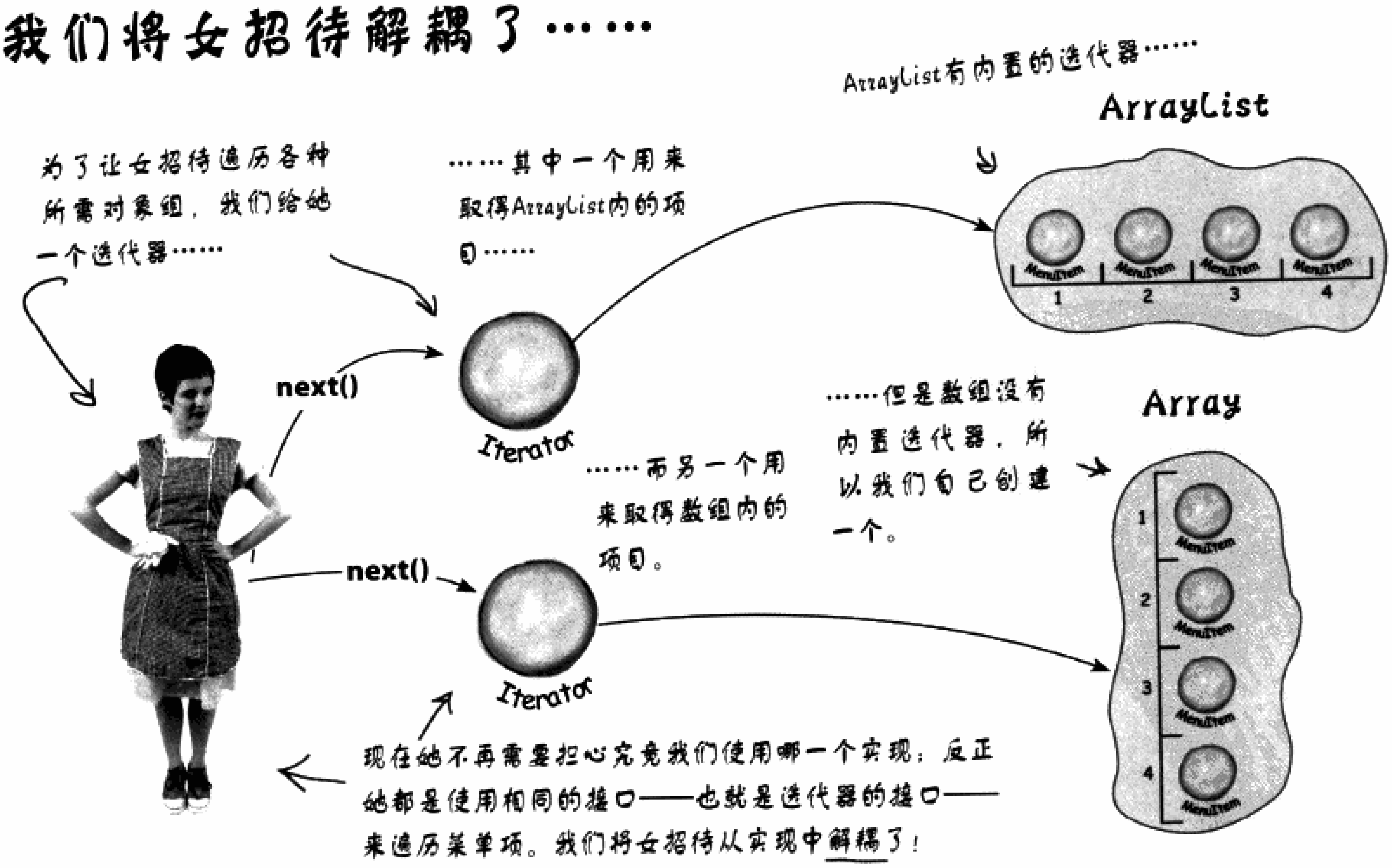
然后是餐厅菜单。

最后是咖啡厅菜单。遍历代码是相同的。

我们做了什么？



我们将女招待解耦了……



……我们让女招待更具有扩展性

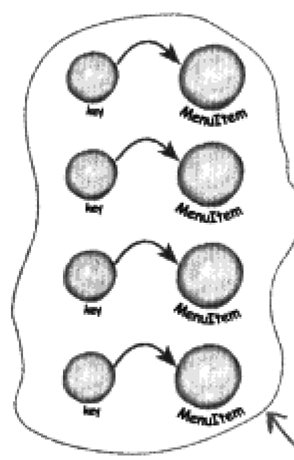
通过赋予她一个迭代器，我们将她从菜单项的实现中解耦了，所以今后我们可以轻易地增加新的菜单。



这对她比较好，因为现在她能够使用相同代码来遍历任意组内的对象。这也对我们比较好，因为我们不用再将实现的细节暴露出去了。

我们很轻易地就加入了另一个菜单项的实现，而且因为我们提供了迭代器，所以女招待知道如何处理这个新的菜单。

Hashtable



为这个Hashtable的值制作一个迭代器。做法很简单：你只要调用values.iterator()，就可以取得一个迭代器。

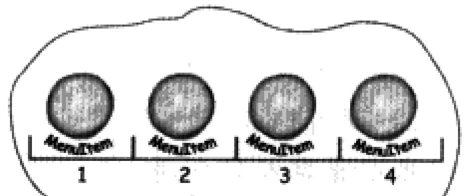
但还有更多！

Java提供你许多的“collection”类（例如：Vector和LinkedList），让你能够存取一群对象。

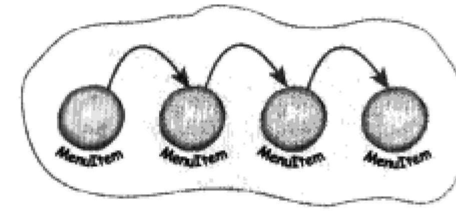
它们具有不同的接口。

但尽管如此，几乎这些类都会提供方法让我们获得迭代器。

Vector



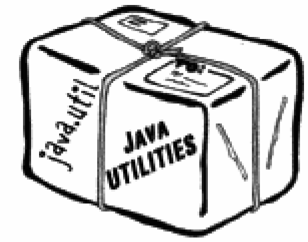
LinkedList



……还有更多！

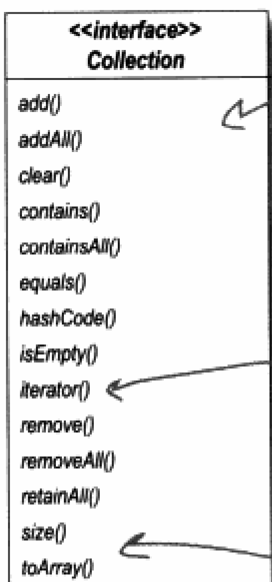
而如果他们不支持迭代器的话，也没关系，因为现在你已经知道如何自己动手创建一个迭代器了。

迭代器与集合



我们所使用的这些类都属于Java Collection Framework的一部分。这里所谓的“framework”（框架）指的是一群类和接口，其中包括了ArrayList、Vector、LinkedList、Stack和PriorityQueue。这些类都实现了java.util.Collection接口。这个接口包含了许多有用的方法，可以操纵一群对象。

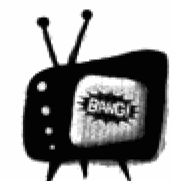
让我们快速地浏览这个接口：



这里有许多好东西。你可以从集合中新增或删除元素，而无需知道这个集合是如何实现的。

这是我们的老朋友，iterator()方法。利用这个方法，你可以取得任意类的迭代器，该迭代器实现了Iterator接口。

其他的方法，包括：size()，可以取得元素的个数，而toArray()用来将集合转成数组。



注意！

Hashtable对于迭代器的支持是“间接的”。当我们在实现咖啡厅菜单的时候，你可以从中取得一个迭代器，但是这个迭代器不是直接从Hashtable取出，而是由Hashtable的value取出的。仔细想想，这很有道理：Hashtable内部存储了两组对象：key和value。如果我们想要遍历value，当然是要先从Hashtable取得value，然后再取得迭代器。



Java 5的迭代器和集合



Java 5包含一种新形式的for语句，称为for/in。这可以让你在一个集合或者一个数组中遍历，而且不需要显式创建迭代器。

想使用for/in，语法是这样的：

在集合中的每个对象之间反复遍历。

每次到循环的最后，obj会被赋值为集合中的下一个元素。

```
for (Object obj: collection) {
        ...
}
```

下面是利用for/in遍历ArrayList的例子：

产生一个菜单项的ArrayList。

```
ArrayList items = new ArrayList();
items.add(new MenuItem("Pancakes", "delicious pancakes", true, 1.59);
items.add(new MenuItem("Waffles", "yummy waffles", true, 1.99);
items.add(new MenuItem("Toast", "perfect toast", true, 0.59);

for (MenuItem item: items) {
    System.out.println("Breakfast item: " + item);
}
```

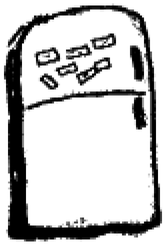
遍历并打印出每一个菜单项



注意！

你需要使用Java 5的泛型（generic）新特性来确保for/in的类型安全。在开始使用generic和for/in之前，请务必继续读下去。

代码帖



厨师们决定午餐的菜单项能交替的改变，也就是说，他们希望在周一、周三、周五和周六提供一些项，然后在周二、周四和周日提供另一些项。

有人已经为新的“轮换”餐厅菜单迭代器书写了代码，但是他们开了一个玩笑，把它打乱并放在冰箱上了。你能够把代码再组织回来吗？其中有些大括号的纸片掉在了地板上，因为太小，不容易捡起来，所以如果有需要的话你可以自己加上大括号。

```
MenuItem menuItem = items[position];
position = position + 2;
return menuItem;
```

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public Object next() {
```

```
}
```

```
public AlternatingDinerMenuIterator(MenuItem[] items)
```

```
this.items = items;
Calendar rightNow = Calendar.getInstance();
position = rightNow.get(calendar.DAY_OF_WEEK) % 2;
```

```
public void remove() {
```

```
implements Iterator
```

```
MenuItem[] items;
int position;
```

```
}
```

```
public class AlternatingDinerMenuIterator
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "Alternating Diner Menu Iterator does not support remove()");
```

```
if (position >= items.length || items[position] == null) {
    return false;
} else {
    return true;
}
```

```
}
```



女招待准备好迎接精采时刻了吗？

我们花了很多时间在女招待上，但还是得承认，程序中调用三次printMenu()，看来实在有点丑。

看清现实，每次我们一有新菜单加入，就必须打开女招待实现并加入更多的代码。这算不算是“违反开放-关闭原则”？

调用createIterator()三次。

```
public void printMenu() {
    Iterator pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator dinerIterator = dinerMenu.createIterator();
    Iterator cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

调用printMenu()三次。

每次我们新增或删除一个菜单，就必须打开这份代码来修改。

这不是女招待的错。对于将她从菜单的实现上解耦和提取遍历动作到迭代器，我们都做得很好。但我们仍然将菜单处理成分离而独立的对象——我们需要一种一起管理它们的方法。

BRAIN POWER

女招待仍然需要调用printMenu()三次，每个菜单一次。你能够想到什么方式将菜单合并以便只需调用一次就可以了？或者只传给女招待一个迭代器，利用这个迭代器就可以遍历所有的菜单？



听起来厨师已有定见，我们试试看：

```
public class Waitress {
    ArrayList menus;

    public Waitress(ArrayList menus) {
        this.menus = menus;
    }

    public void printMenu() {
        Iterator menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = (Menu)menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

现在我们只需要一个菜单ArrayList。

我们遍历菜单，把每个菜单的迭代器传给重载的printMenu()方法。

这里的代码不需要改变。

看起来相当不错，虽然我们失去了菜单的名字，但是可以把名字加进每个菜单中。

正当我们认为这很安全的时候……

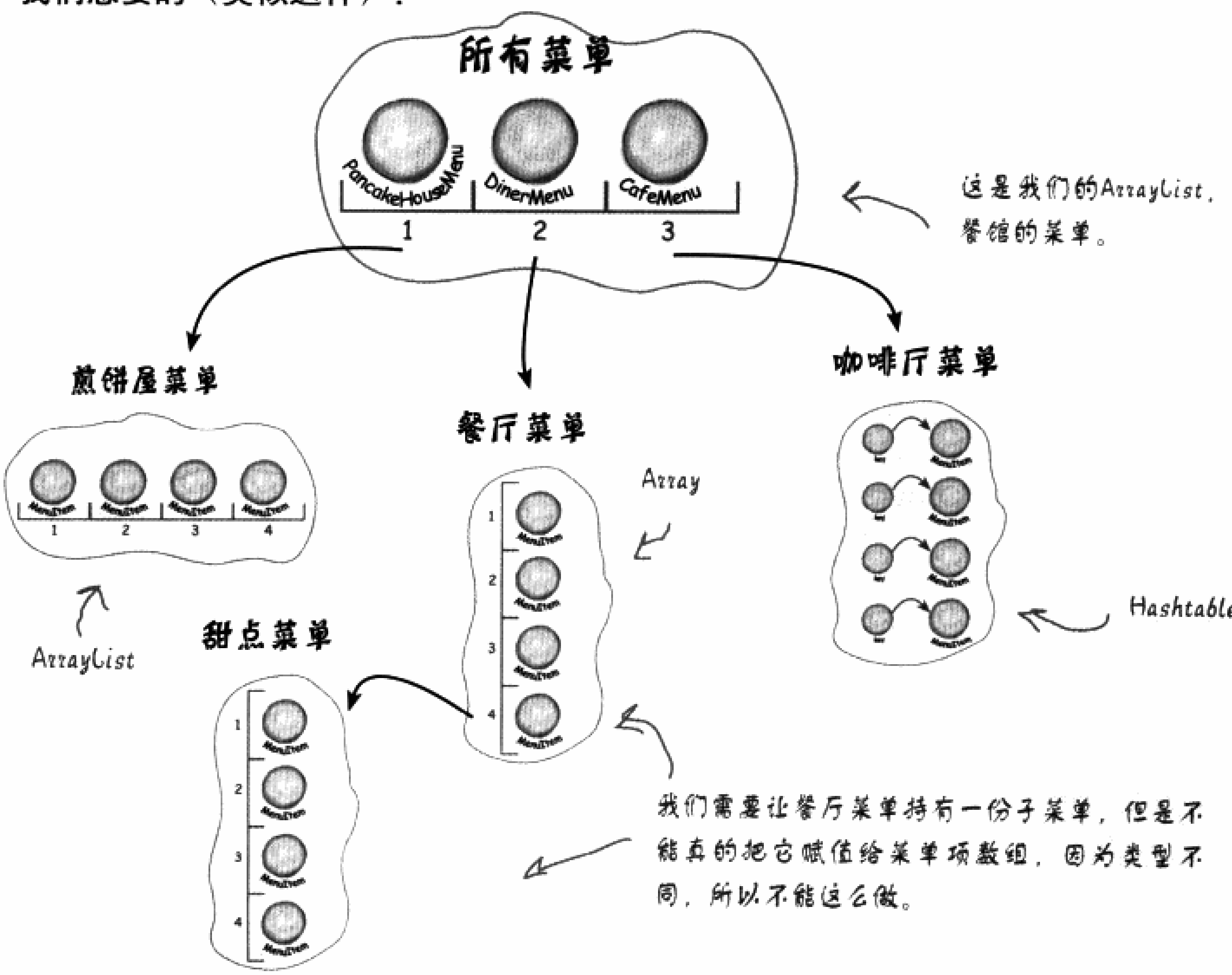
现在他们希望能够加上一份餐后甜点的“子菜单”。

现在怎么办？我们不仅仅要支持多个菜单，甚至还要支持菜单中的菜单。



如果我们能让甜点菜单变成餐厅菜单集合的一个元素，那该有多好。但是根据现在的实现，根本做不到。

我们想要的（类似这样）：



但是这行不通！

我们不能把甜点菜单赋值给菜单项数组。

又要修改了！

我们需要什么？

该是做决策来改写厨师的实现以符合所有菜单（以及子菜单）的需求的时候了。没错，我们要告诉厨师，重新实现他们的菜单已经是不可避免的了。

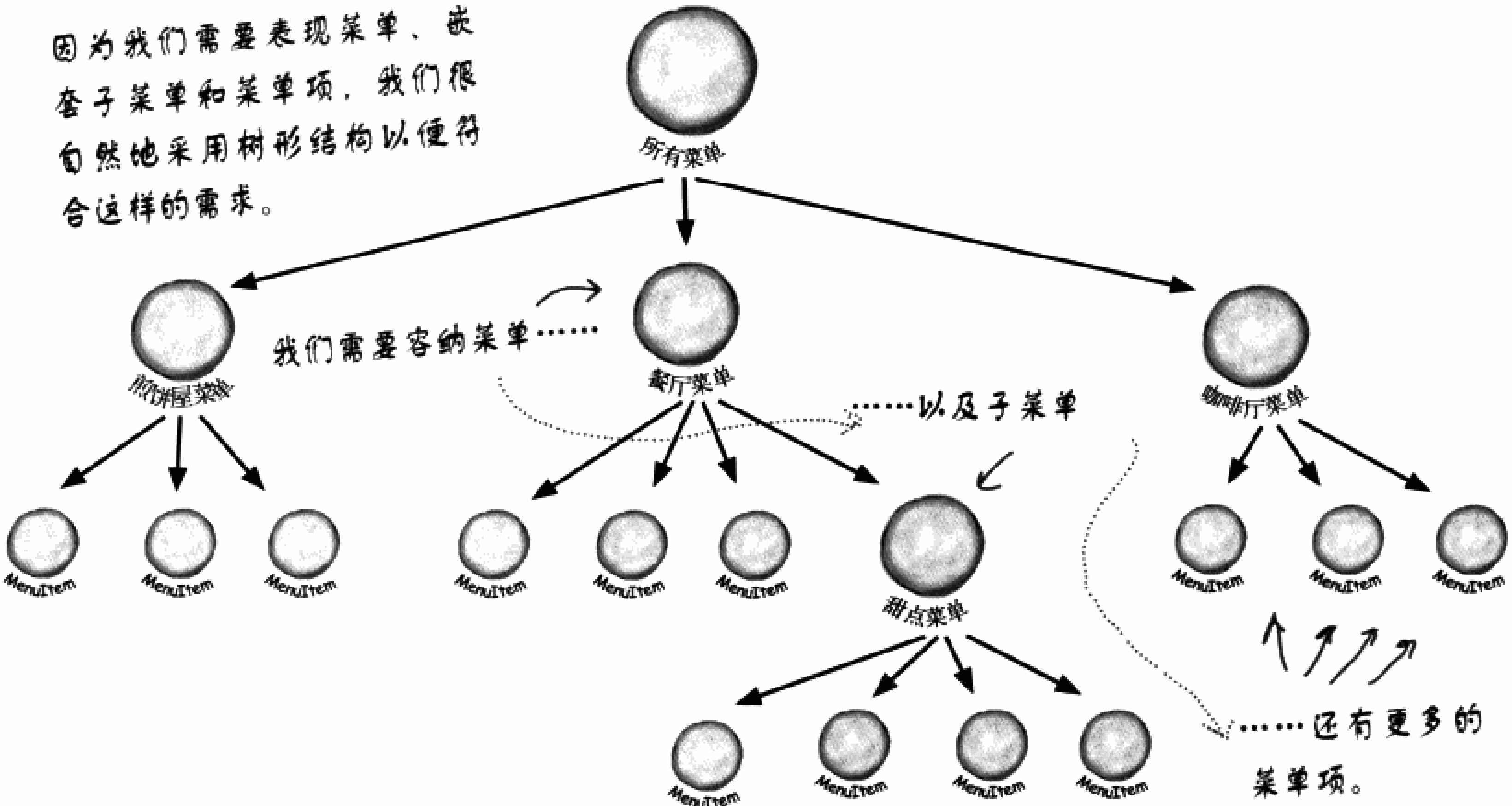
事实是，我们已经到达了一个复杂级别，如果现在不重新设计，就无法容纳未来增加的菜单或子菜单等需求。

所以，在我们的新设计中，真正需要些什么呢？

- 我们需要某种树形结构，可以容纳菜单、子菜单和菜单项。
- 我们需要确定能够在每个菜单的各个项之间游走，而且至少要像现在用迭代器一样方便。
- 我们也需要能够更有弹性地在菜单项之间游走。比方说，可能只需要遍历甜点菜单，或者可以遍历餐厅的整个菜单（包括甜点菜单在内）。

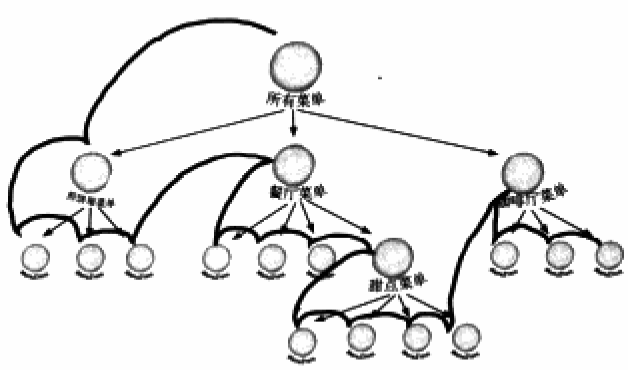


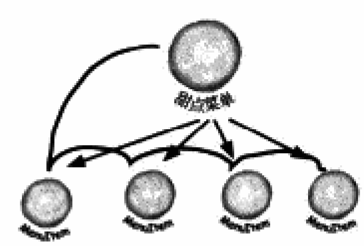
因为我们需要表现菜单、嵌套子菜单和菜单项，我们很自然地采用树形结构以便符合这样的需求。



我们也需要更有弹性的游走方式，比方说，我们可能只需要遍历餐厅的甜点菜单，也可能需要遍历餐厅的全部菜单，包括甜点菜单。

我们仍然需要能在树上的所有项之间游走。



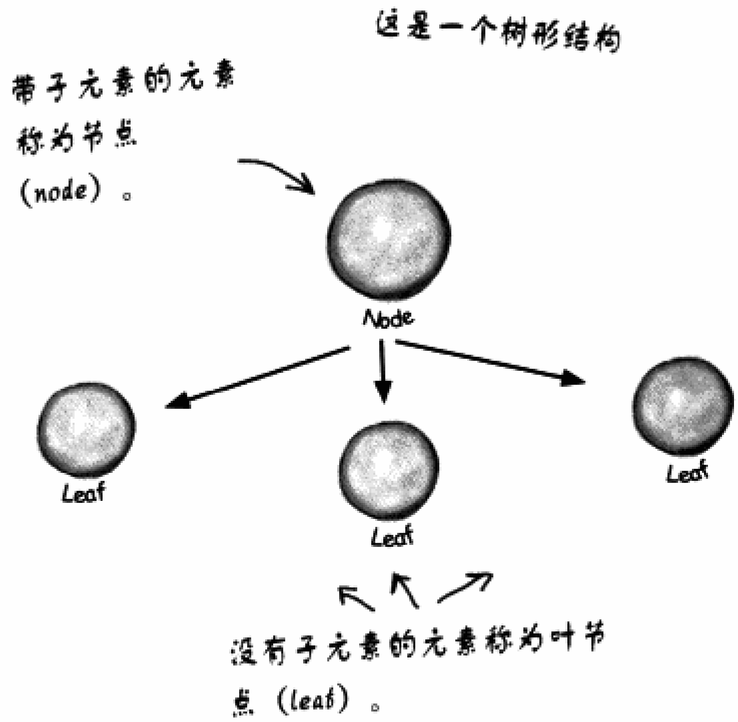


BRAIN POWER

你如何处理这个新的设计需求？在翻页之前请先想一想。

定义组合模式

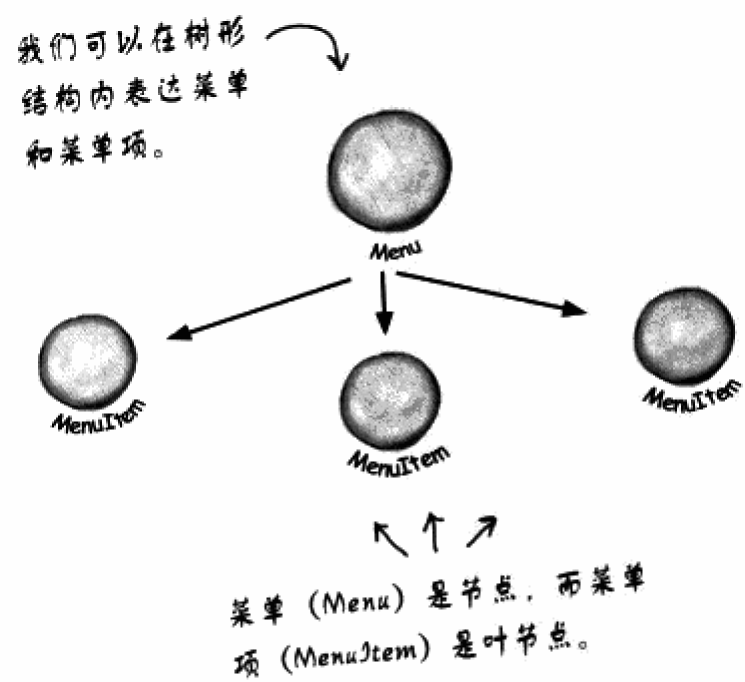
没错，我们要介绍另一个模式解决这个难题。我们并没有放弃迭代器——它仍然是我们解决方案中的一部分——然而，管理菜单的问题已经到了一个迭代器无法解决的新维度。所以，我们将倒退几步，改用组合模式（Composite Pattern）来实现这一部分。



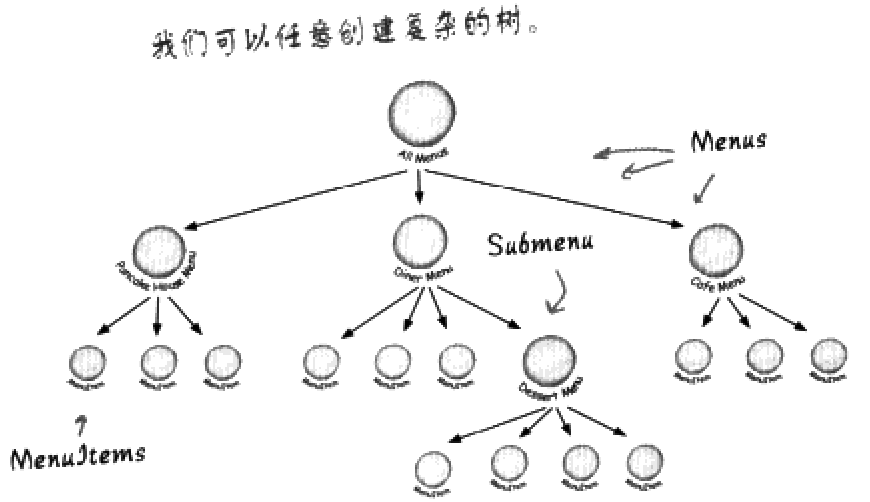
对于这个模式，我们不打算深入探讨，只在这里提出它的正式定义：

> **组合模式**允许你将对象组合成树形结构来表现“整体/部分”层次结构。组合能让客户以一致的方式处理个别对象以及对象组合。

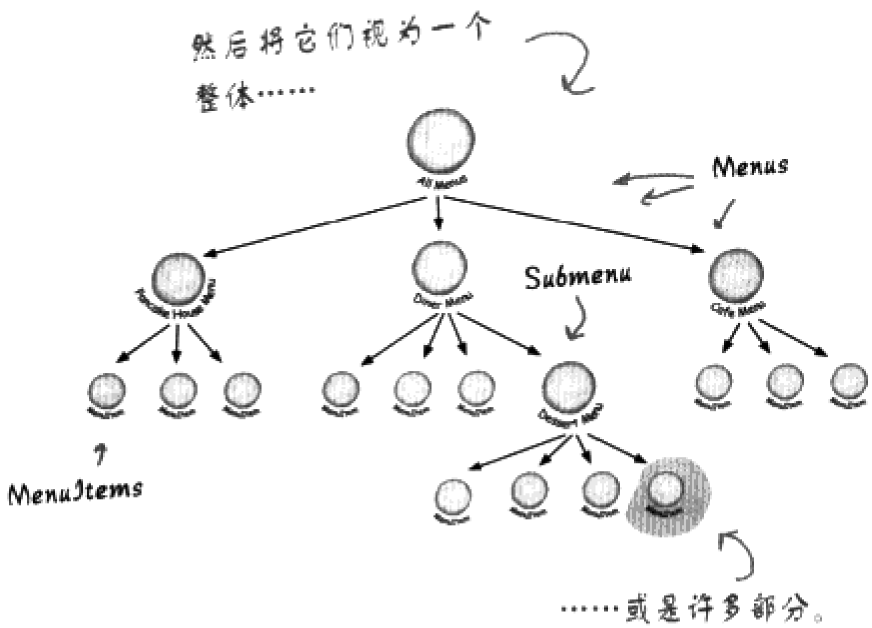
让我们以菜单为例思考这一切：这个模式能够创建一个树形结构，在同一个结构中处理嵌套菜单和菜单项组。通过将菜单和项放在相同的结构中，我们创建了一个“整体/部分”层次结构，即由菜单和菜单项组成的对象树。但是可以将它视为一个整体，像是一个丰富的大菜单“译注：uberr来自德文，相当于英文的over”。



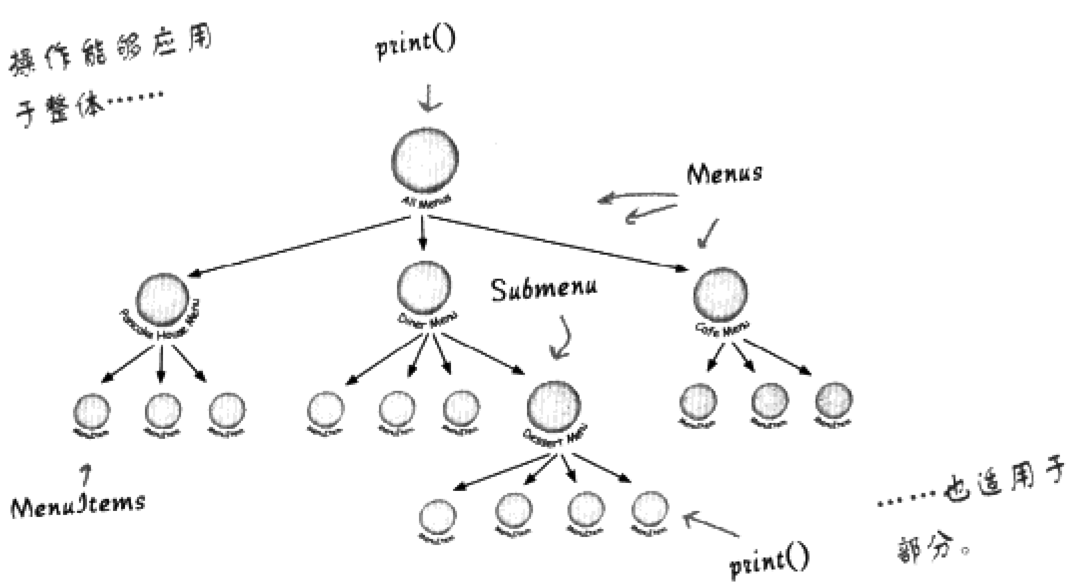
一旦有了丰富的大菜单，我们就可以使用这个模式来“统一处理个别对象和组合对象”。这意味着什么？它意味着，如果我们有了一个树形结构的菜单、子菜单和可能还带有菜单项的子菜单，那么任何一个菜单都是一种“组合”。因为它既可以包含其他菜单，也可以包含菜单项。个别对象只是菜单项——并未持有其他对象。就像你将看到的，使用一个遵照组合模式的设计，让我们能够写出简单的代码，就能够对整个菜单结构应用相同的操作（例如打印！）。

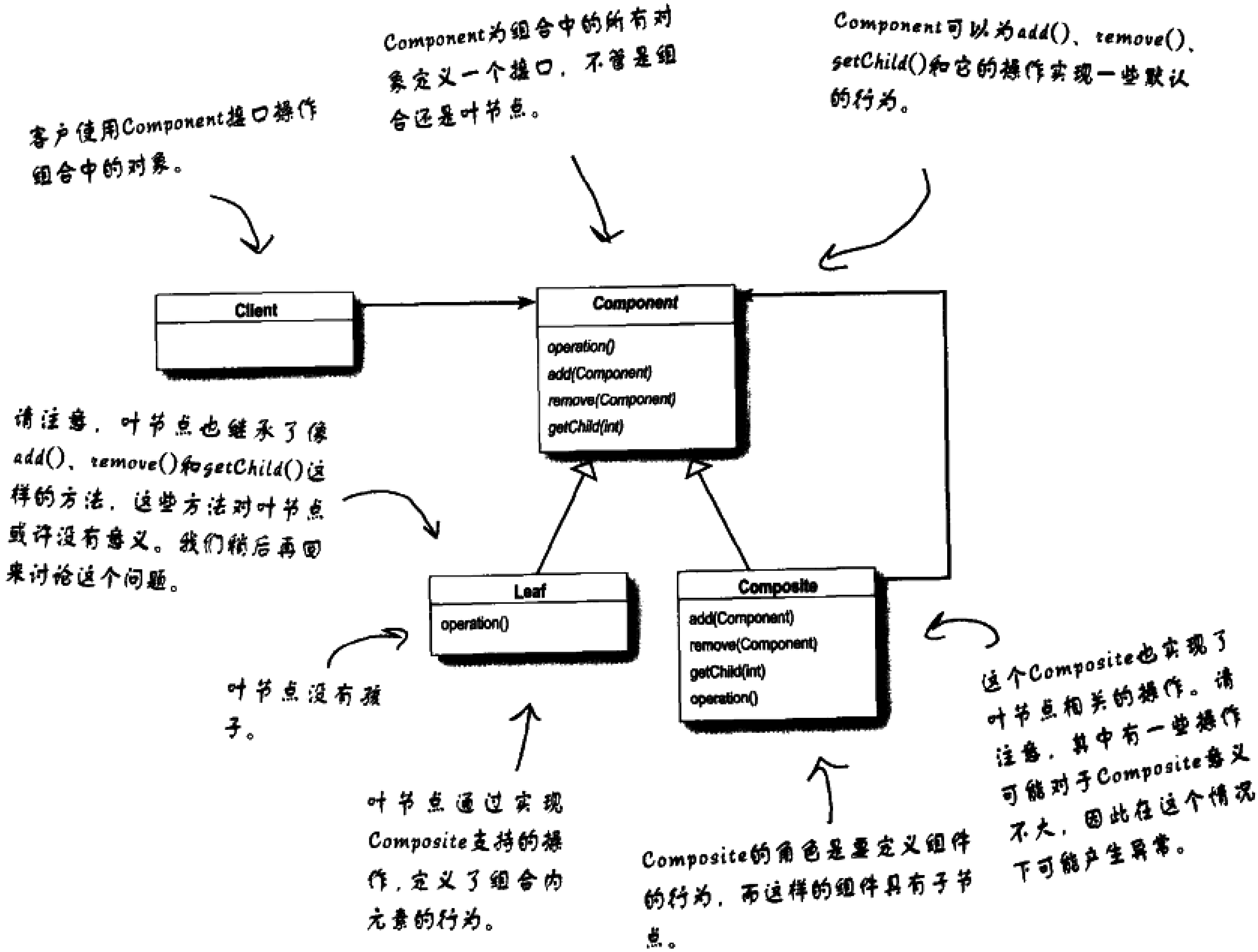


组合模式让我们能用树形方式创建对象的结构，树里面包含了组合以及个别的对象。



使用组合结构，我们能把相同的操作应用在组合和个别对象上。换句话说，在大多数情况下，我们可以忽略对象组合和个别对象之间的差别。





there are no Dumb Questions

问： 组件、组合、树？我被搞混了。

答： 组合包含组件。组件有两种：组合与叶节点元素。听起来象递归是不是？组合持有一群孩子，这些孩子可以是别的组合或者叶节点元素。

当你用这种方式组织数据的时候，最终会得到树形结构（正确的说法是由上而下的树形结构），根部是一个组合，而组合的分支逐渐往下延伸，直到叶节点为止。

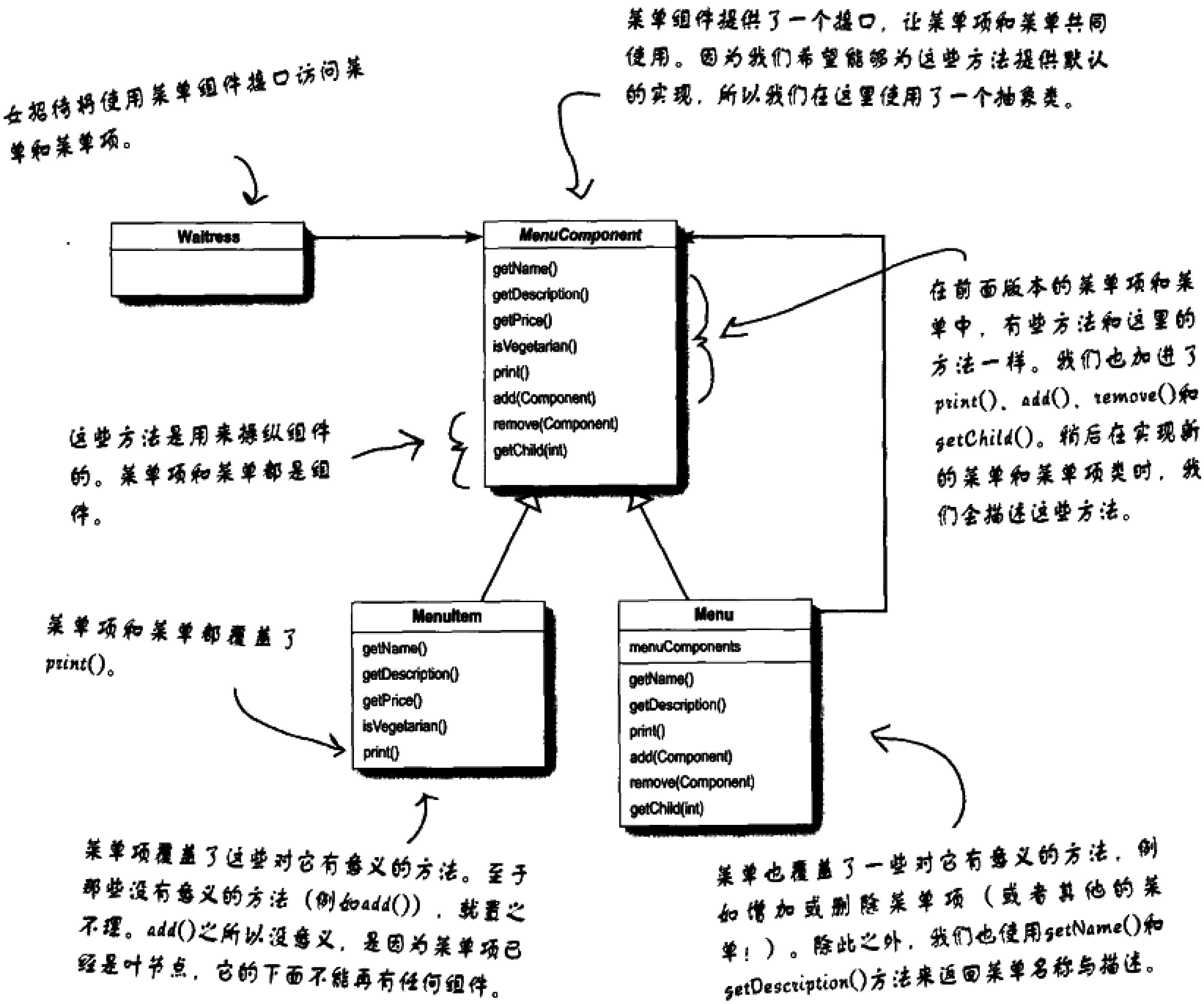
问： 这和迭代器有什么关系？

答： 别忘了，我们现在采取了一个新方法，打算用新的方案——组合模式，来重新实现菜单。所以不要认为迭代器和组合之间有什么神奇的转换。我们可以说，这两者可以合作无间，你很快就会看到我们可以在组合的实现中使用迭代器，而且做法还不只一种。

利用组合设计菜单

我们要如何在菜单上应用组合模式呢？一开始，我们需要创建一个组件接口来作为菜单和菜单项的共同接口，让我们能够用统一的做法来处理菜单和菜单项。换句话说，我们可以针对菜单或菜单项调用相同的方法。

现在，对于菜单或菜单项来说，有些方法可能不太恰当。但我们可以处理这个问题，等一下就会这么做。至于现在，让我们从头来看看如何让菜单能够符合组合模式的结构：



实现菜单组件

好了，我们要开始编写菜单组件的抽象类；请记住，菜单组件的角色是为叶节点和组合节点提供一个共同的接口。现在你可能想问："那么菜单组件不就扮演了两个角色吗？"可能是这样的，我们稍后再讨论这一点。然而，目前我们要为这些方法提供默认的实现，这样，如果菜单项（叶节点）或者菜单（组合）不想实现某些方法的时候（例如叶节点不想实现getChild()方法），就可以不实现这些方法。

所有的组件都必须实现MenuComponent接口；然而，叶节点和组合节点的角色不同，所以有些方法可能并不适合某种节点。面对这种情况，有时候，你最好是抛出运行时异常

因为有些方法只对菜单项有意义，而有些则只对菜单有意义，默认实现是抛出UnsupportedOperationException异常。这样，如果菜单项或菜单不支持某个操作，他们就不需做任何事情，直接继承默认实现就可以了。

MenuComponent对每个方法都提供默认的实现。

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

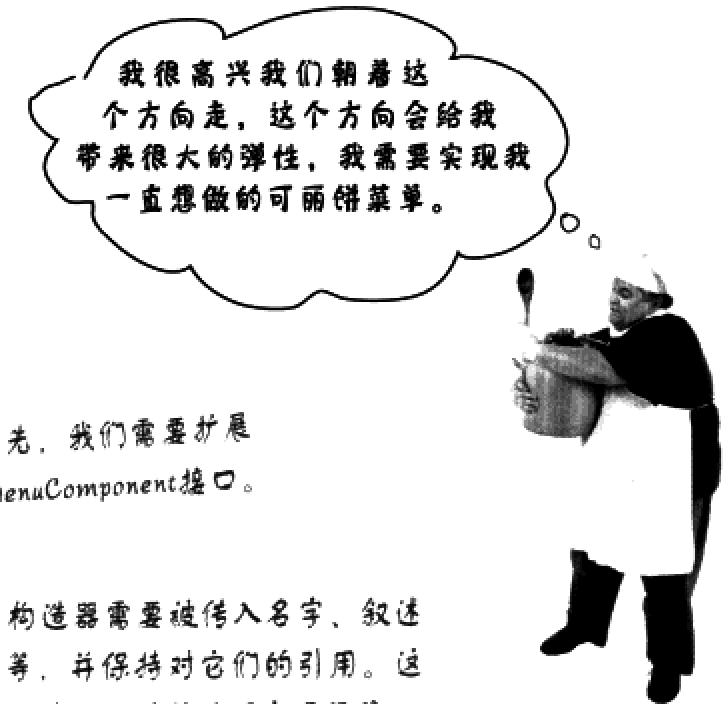
    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

我们把"组合"方法组织在一起，即新增、删除和取得菜单组件。

这些是"操作"方法；它们被菜单项使用，其中有一些也可用在菜单上，再过几页你就会在菜单代码中看到了。

print()是一个"操作"方法，这个方法同时被菜单和菜单项所实现，但我们还是在这里提供了默认的操作。

实现菜单项



好了，让我们来看菜单项类。别忘了，这是组合类图里的叶类，它实现组合内元素的行为。

```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

首先，我们需要扩展MenuComponent接口。

构造器需要被传入名字、叙述等，并保持对它们的引用。这和我们旧的菜单项实现很像。

这是我们的getter方法，和之前的实现一样。

这和之前的实现不一样，在MenuComponet类里我们覆盖了print()方法。对菜单项来说，此方法会打印出完整的菜单项条目，包括：名字、描述、价格以及是否为素食。

实现组合菜单

我们已经有了菜单项，还需要组合类，就是我们叫做菜单的。别忘了，此组合类可以持有菜单项或其他菜单。有一些方法并未在MenuComopnent类中实现，比如getPrice()和isVegertarian()，因为这些方法对菜单而言并没多大意义。

菜单和菜单项一样，都是MenuComponent。

菜单可以有任意数目的孩子，这些孩子都必须属于MenuComponent类型，我们使用内部的ArrayList记录它们。

这和我们之前的实现不一样，我们将给每个菜单一个名字和一个描述。以前，每个菜单的类名称就是此菜单的名字。

我们在这里将菜单项和其他菜单加入到菜单中。因为菜单和菜单项都是MenuComponent，所以我们只需用一个方法就可以两者兼顾。

同样的道理，也可以删除或取得某个MenuComponent。

这是用来取得名字和描述的getter方法。

请注意，我们并未覆盖getPrice()或isVegertarian()，因为这些方法对Menu来说并没有意义（虽然你可能认为isVegertarian()有意义）。如果有人试着在Menu上调用这些方法，就会得到UnsupportedOperationException异常。

为了打印出菜单，我们打印菜单的名称和描述。

```java
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");
    }
}
```

> 等一下，我不了解print()的实现。原本我以为应该能够在组合和叶节点上应用相同的操作。如果我用这种实现对组合节点应用print()，所得到的竟然只是一个简单的菜单名字和描述，而不是完整地打印出组合内的每个项。



说得好。因为菜单是一个组合，包含了菜单项和其他的菜单，所以它的print()应该打印出它所包含的一切。如果它不这么做，我们就必须遍历整个组合的每个节点，然后将每一项打印出来。这么一来，也就失去了使用组合结构的意义。

想要正确地实现print()其实很容易，因为我们可以让每个组件打印自己，这种递归方式简直美妙极了，赶快来看看吧：

修正print()方法

```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // 构造器代码在这里

    // 其他的方法在这里

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

我们所要做的只是改变print()方法，能让它不仅打印出菜单本身的信息，也打印出菜单内所有组件的内容：其他菜单和菜单项。

看吧！我们用了迭代器。用它遍历所有菜单组件……遍历过程中，可能遇到其他菜单，或者是遇到菜单项。由于菜单和菜单项都实现了print()，那我们只要调用print()即可。

请注意：在遍历期间，如果遇到另一个菜单对象，它的print()方法会开始另一个遍历，依次类推。

测试前的准备工作……

差不多可以测试了，但是在开始测试之前，我们必须更新女招待的代码——毕竟她是菜单的主要客户：

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

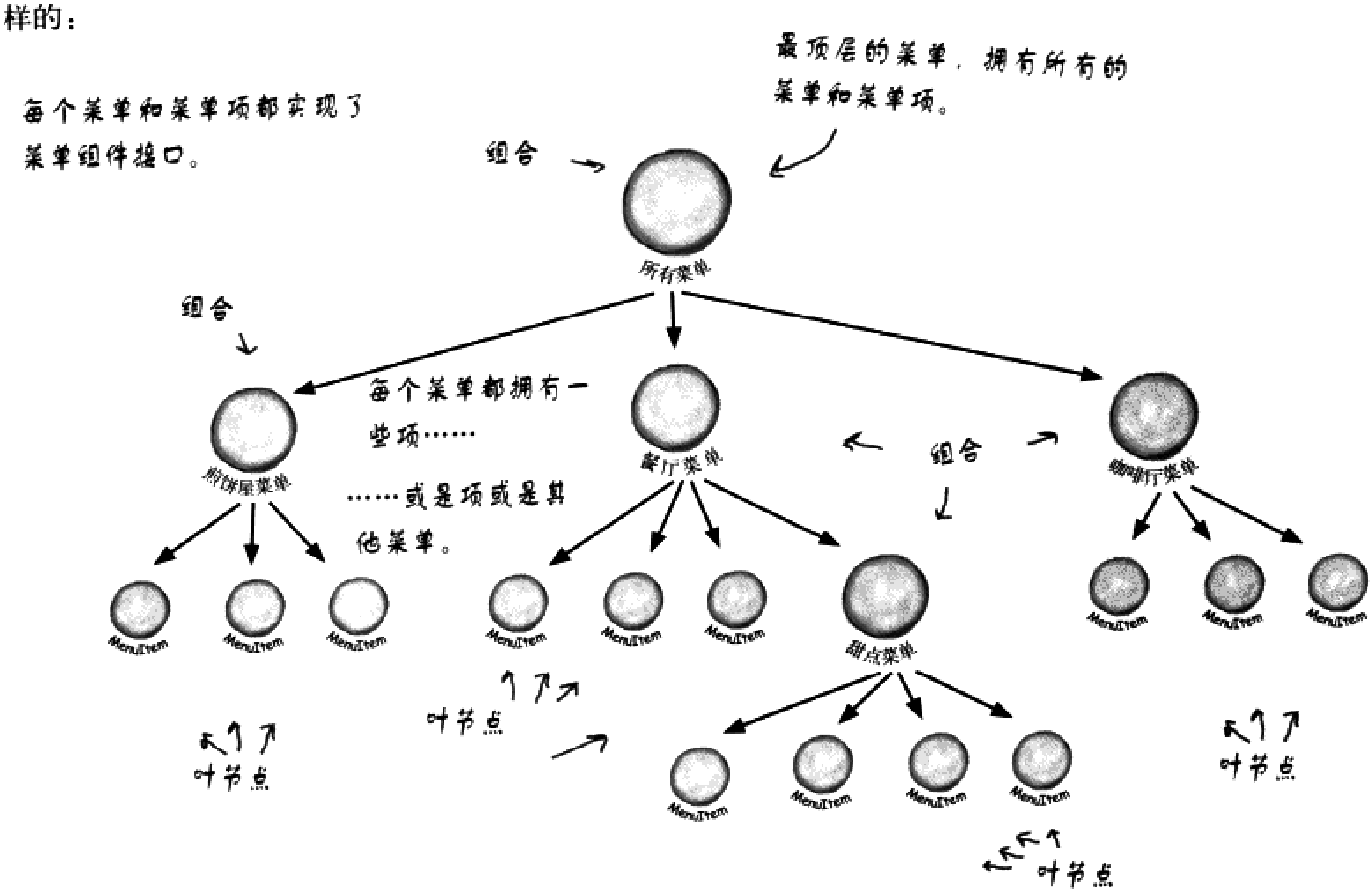
    public void printMenu() {
        allMenus.print();
    }
}
```

好！女招待的代码真的变得很简单。现在我们只要将最顶层的菜单组件交给她就可以了，最顶层菜单包含其他所有菜单，我们称之为allMenus。

她只需要调用最顶层菜单的print()，就可以打印整个菜单层次，包括所有菜单及所有菜单项。

这个女招待会变得很快乐。

好了，在开始测试前，还剩下最后一件事。让我们了解一下，在运行时菜单组合是什么样的：



编写测试程序……

好了，现在要写一个测试程序。和以前的版本不同，我们这个版本要在测试程序中处理所有菜单的创建。我们可以请每位厨师交出他的新菜单，但是让我们先将这一切测试完毕。代码如下：

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // 在这里加入菜单项

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla ice cream",
            true,
            1.59));

        // 在这里加入更多菜单项

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

先创建所有的菜单对象。

我们需要一个最顶层的菜单，将它称为allMenus。

我们使用组合的add()方法，将每个菜单都加入到顶层菜单allMenus中。

现在我们需要加上所有的菜单项，这是一个例子，至于其他的菜单项，请看完整的源码。

然后我们也在菜单中加入另一个菜单。由于菜单和菜单项都是MenuComponent，所以菜单可以顺利地被加入。

在甜点菜单上加了苹果派……

一旦我们将整个菜单层次构造完毕，把它整个交给女招待，你会发现，女招待要将整份菜单打印出来，简直就是易如反掌。

执行结果……

请注意：这份执行结果基于完整的源码。

```
File Edit Window Help GreenEggs&Spam
% java MenuTestDrive

ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

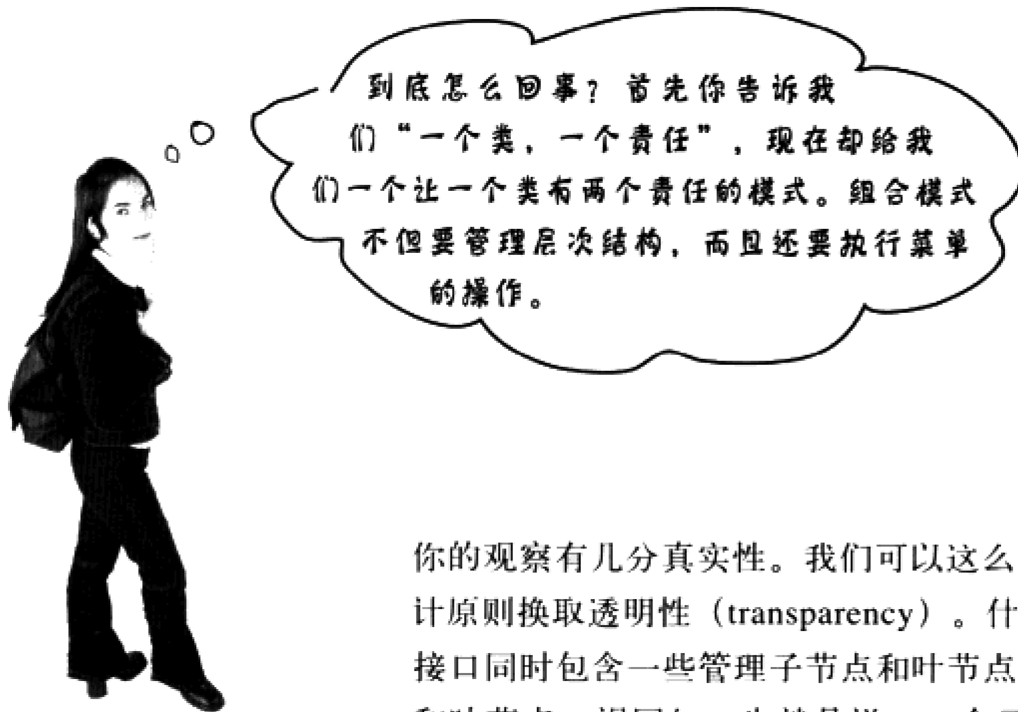
DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
     -- A hot dog, with saurkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DESSERT MENU, Dessert of course!
---------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla ice cream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
---------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

我们的菜单都在这儿……只要调用最顶层菜单的print()，就可以打印出一切。

当打印所有餐厅菜单组件时，新的甜点菜单也一起打印出来。



你的观察有几分真实性。我们可以这么说，组合模式以单一责任设计原则换取透明性（transparency）。什么是透明性？通过让组件的接口同时包含一些管理子节点和叶节点的操作，客户就可以将组合和叶节点一视同仁。也就是说，一个元素究竟是组合还是叶节点，对客户是透明的。

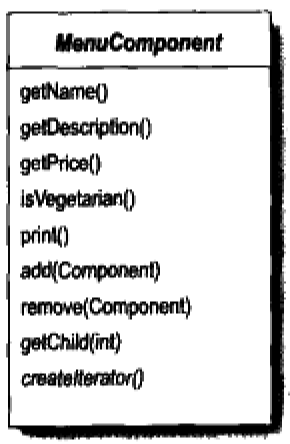
现在，我们在MenuComponent类中同时具有两种类型的操作。因为客户有机会对一个元素做一些不恰当或是没有意义的操作（例如试图把菜单添加到菜单项），所以我们失去了一些"安全性"。这是设计上的抉择；我们当然也可以采用另一种方向的设计，将责任区分开来放在不同的接口中。这么一来，设计上就比较安全，但我们也因此失去了透明性，客户的代码将必须用条件语句和instanceof操作符处理不同类型的节点。

所以，回到你的问题，这是一个很典型的折衷案例。尽管我们受到设计原则的指导，但是，我们总是需要观察某原则对我们的设计所造成的影响。有时候，我们会故意做一些看似违反原则的事情。然而，在某些例子中，这是观点的问题；比方说，让管理孩子的操作（例如add()、remove()、getChild()）出现在叶节点中，似乎很不恰当，但是换个视角来看，你可以把叶节点视为没有孩子的节点。

闪回到迭代器

几页前，我们答应过会告诉你怎样用组合来使用迭代器。我们其实已经在print()方法内部的实现中使用了迭代器，除此之外，如果女招待需要，我们也能让她使用迭代器遍历整个组合。比方说，女招待可能想要游走整个菜单，挑出素食项。

想要实现一个组合迭代器，让我们为每个组件都加上createIterator()方法。从抽象的MenuComponent类开始下手：



我们在MenuComponent中加入一个createIterator()方法。这意味着，每个菜单和菜单项都必须实现这个方法。也意味着，对一个组合调用createIterator()方法，将会应用于该组合的所有孩子。

现在我们需要在菜单和菜单项类中实现这个方法：

这里使用一个新的，被称为CompositeIterator的迭代器。这个迭代器知道如何遍历任何组合。我们将目前组合的迭代器传入它的构造器。

```
public class Menu extends MenuComponent {

    // 其他部分的代码不需要修改

    public Iterator createIterator() {
        return new CompositeIterator(menuComponents.iterator());
    }

}
```

```
public class MenuItem extends MenuComponent {

    // 其他部分的代码不需要修改

    public Iterator createIterator() {
        return new NullIterator();
    }

}
```

现在轮到菜单项……
天呀！什么是NullIterator？
再过两页你就会知道了。

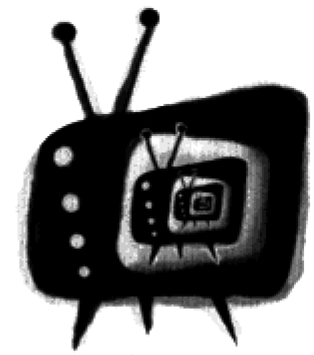
组合迭代器

这个CompositeIterator是一个不可小觑的迭代器。它的工作是遍历组件内的菜单项，而且确保所有的子菜单（以及子子菜单……）都被包括进来。

代码是下面这样的。请注意，代码虽然不多，但是不见得容易理解。

注意：

跟着我默念“递归是我的朋友，递归是我的朋友……”



当心，前面是递归区！

跟所有的迭代器一样，我们实现了java.util.Iterator接口。

```
import java.util.*;

public class CompositeIterator implements Iterator {
    Stack stack = new Stack();

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

将我们要遍历的顶层组合的迭代器传入。我们把它抛进一个堆栈数据结构中。

好了，当客户想要取得下一个元素的时候，我们先调用hasNext()来确定是否还有下一个。

如果还有下一个元素，我们就从堆栈中取出目前的迭代器，然后取得它的下一个元素。

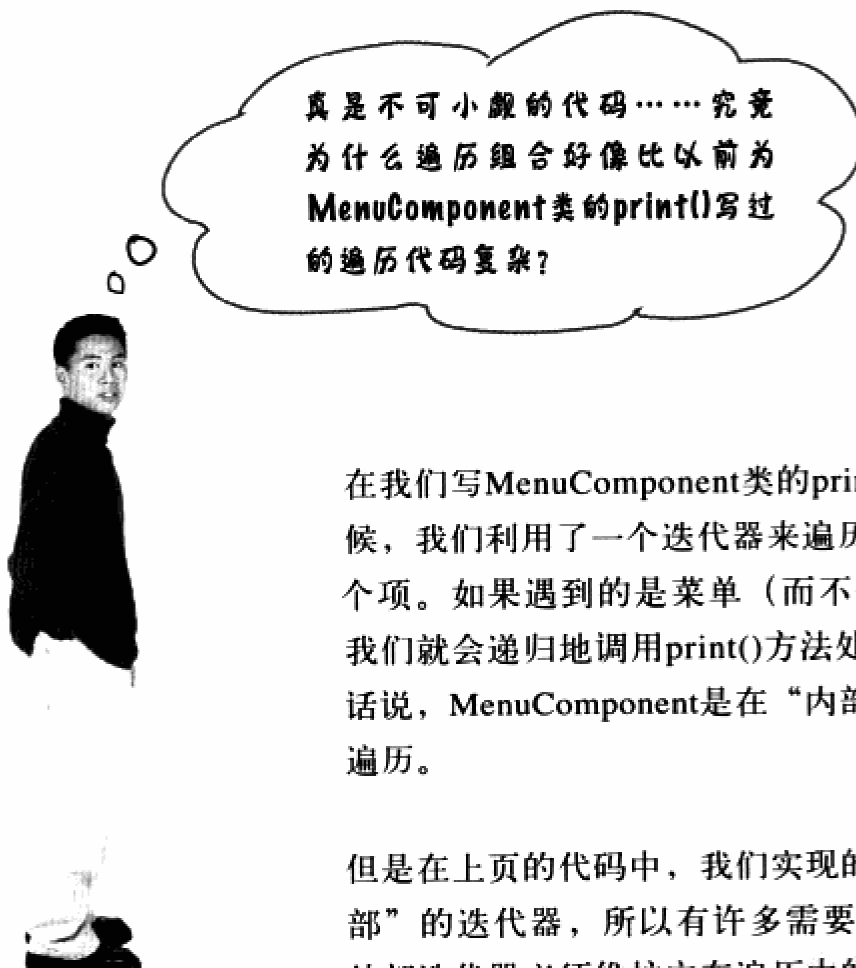
如果元素是一个菜单，我们有了另一个需要被包含进遍历中的组合，所以我们将它丢进堆栈中。不管是不是菜单，我们都返回该组件。

想要知道是否还有下一个元素，我们检查堆栈是否被清空；如果已经空了，就表示没有下一个元素了。

否则，我们就从堆栈的顶层中取出迭代器，看看是否还有下一个元素。如果它没有元素，我们将它弹出堆栈，然后递归地调用hasNext()。

否则，表示还有下一个元素，我们返回true。

我们不支持删除，这里只有遍历。



在我们写MenuComponent类的print()方法的时候，我们利用了一个迭代器来遍历组件内的每个项。如果遇到的是菜单（而不是菜单项），我们就会递归地调用print()方法处理它。换句话说，MenuComponent是在“内部”自行处理遍历。

但是在上页的代码中，我们实现的是一个“外部”的迭代器，所以有许多需要追踪的事情。外部迭代器必须维护它在遍历中的位置，以便外部客户可以通过调用hasNext()和next()来驱动遍历。在这个例子中，我们的代码也必须维护组合递归结构的位置。这也就是为什么当我们在组合层次结构中上上下下时，使用堆栈来维护我们的位置。

BRAIN POWER

针对菜单和菜单项绘制一张图。然后假装你是CompositeIterator，而你的工作是处理对hasNext()和next()的调用。请在下面的代码执行过程中，追踪CompositeIterator的足迹。

```
public void testCompositeIterator(MenuComponent component) {
        CompositeIterator iterator = new CompositeIterator(component.iterator);

        while(iterator.hasNext()) {
                MenuComponent component = iterator.next();
        }
}
```

空迭代器

到底什么是空迭代器（NullIterator）呢？这么说好了：菜单项内没什么可以遍历的，对吧？那么我们要如何实现菜单项的createIterator()方法呢？有两种选择：

请注意：这是空对象"设计模式"的另一个例子。

选择一：

返回null

我们可以让createIterator()方法返回null，但是如果这么做，我们的客户代码就需要条件语句来判断返回值是否为null。

选择二：

返回一个迭代器，而这个迭代器的hasNext()永远返回false

这似乎是一个更好的方案。我们依然可以返回一个迭代器，客户不用再担心返回值是否为null。我们等于是创建了一个迭代器，其作用是"没作用"。

当然第二个选择看起来比较好。让我们称它为空迭代器，下面是它的实现：

这是你所看过最懒的迭代器，什么事情都不做。

```
import java.util.Iterator;

public class NullIterator implements Iterator {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

当next()被调用时，返回null。

最重要的，当hasNext()被调用时，永远返回false。

空迭代器当然不支持remove。

给我素食菜单

现在，我们已经有一种方式可以遍历菜单的每个项了。让我们为女招待加上一个可以确切地告诉我们哪些项目是素食的方法。

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        Iterator iterator = allMenus.createIterator();
        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                    (MenuComponent)iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

printVegetarinMenu()方法取得allMenu's的组合并得到它的迭代器来作为我们的CompositeIterator。

遍历组合内的每个元素。

调用每个元素的isVegetarian()方法，如果为true，就调用它的print()方法。

只有菜单项的print()方法可以被调用，绝对不能调用菜单（组合）的print()方法。你能说出原因吗？

我们在菜单上实现isVegetarian()方法，让它永远抛出异常。如果异常果真发生了，我们就捕捉这个异常，然后继续遍历。

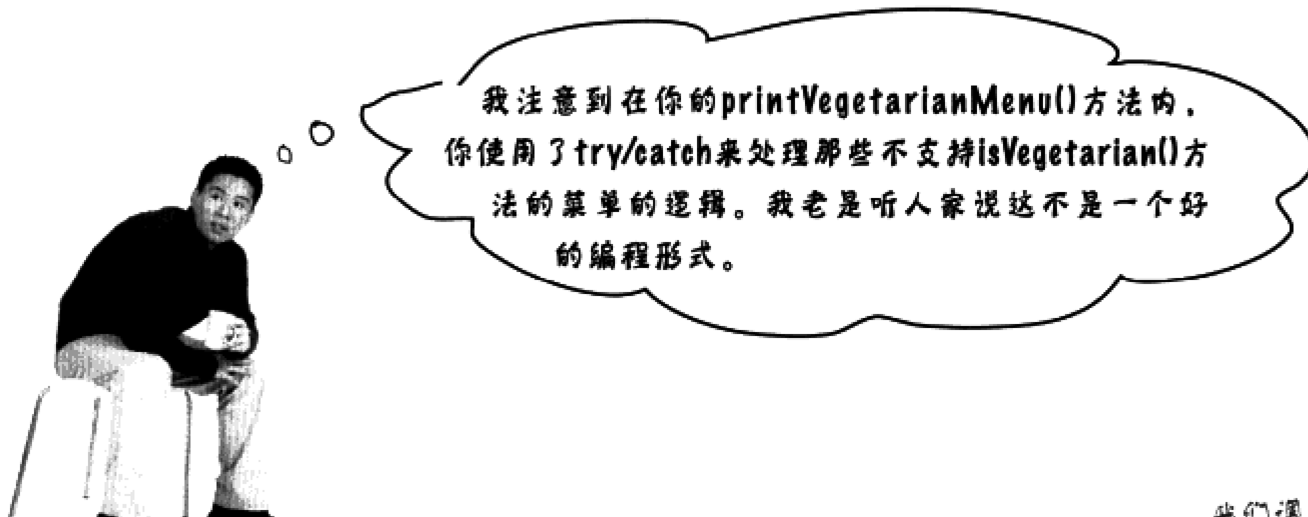
迭代器和组合凑在一起的魔力……

我们可是费了好大一番工夫才走到这里的。现在我们已经有了一个总菜单结构，可以应对未来餐饮帝国的成长需求了。现在让我们坐下休息一会儿，顺便点些素食来吃吧：

```
File Edit Window Help HaveUhuggedYurIteratorToday?
% java MenuTestDrive

VEGETARIAN MENU
----
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla ice cream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla ice cream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

素食菜单内包含了每个菜单内的素食项。



你说的是这个吧：

```
try {
    if (menuComponent.isVegetarian()) {
        menuComponent.print();
    }
} catch (UnsupportedOperationException) {}
```

我们调用全部MenuComponent的isVegetarian()方法，但是Menu会抛出一个异常，因为它们不支持这个操作。

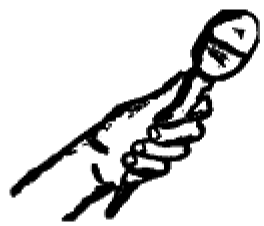
如果菜单组件不支持这个操作，那我们就对这个异常置之不理。

一般来说，我们同意你的看法；try/catch是一种错误处理的方法，而不是程序逻辑的方法。如果不这么做，我们还有哪些选择呢？我们可以在调用isVegetarian()方法之前，用instanceof来检查菜单组件的运行时类型，以确定它是菜单项。但是这么做，我们就会因为无法统一处理菜单和菜单项而失去透明性。

我们也可以改写Menu的isVegetarian()方法，让它返回false。这提供了一个简单的解决方案，同时也保持了透明性。

我们的解决方案是为了要清楚地表示我们的想法。我们真正想要传达的是：isVegetarian()是Menu没有支持的操作（这和说isVegetarian()是false意义不等同）。这样的做法也允许后来人去为Menu实现一个合理的isVegetarian()方法，而我们不必为此再修改这里的代码了。

这是我们的说法，而且我们坚持这么做。



模式告白

本周访问：
组合模式，我们要讨论他在实现上的问题

HeadFirst：我们今天晚上的谈话来宾是组合模式。组合，请向大家介绍一下你自己。

组合：好的……当你有数个对象的集合，它们彼此之间有“整体/部分”的关系，并且你想用一致的方式对待这些对象时，你就需要我。

HeadFirst：好了，让我们从这里深入……你所谓的“整体/部分”关系，指的是什么？

组合：就拿图形用户界面来说，你经常会看到一个顶层的组件（像是Frame或Panel）包含着其他组件（像菜单、文字面板、滚动条、按钮）所以你的GUI包含了若干部分，但是当你显示它的时候，你认为它是一个整体。你告诉顶层的组件显示，然后就放手不管，由顶层组件负责显示所有相关的部分。

我们称这种包含其他组件的组件为组合对象，而称没有包含其他组件的组件为叶节点对象。

HeadFirst：至于你所谓的“用一致的方式对待”所有的对象，又是什么意思？是不是说组合和叶节点之间具有共同的方法可以调用？

组合：没错。我可以叫组合对象显示或是叫叶节点对象显示，他们会各自做出正确的事情。组合对象会叫它所有的组件显示。

HeadFirst：这意味着每一个对象都有相同的接口。万一组合中有些对象的行为不太一样，怎么办？

组合：这个嘛，为了要保持透明性，组合内所有的对象都必须实现相同的接口，否则客户就必须操心哪个对象是用哪个接口，这就失去了组合模式的意义。很明显的，这也意味着有些对象具备一些没有意义的方法调用。

HeadFirst：那怎么办？

组合：有些方式可以处理这一点。有时候你可以让这样的方法不做事，或者返回null值或false。至于挑哪一种方式，就看哪一种在你的应用中比较合乎逻辑。

有时候，你可能想要采取更激烈一点的手法，直接抛出异常。当然，客户就要愿意多做一些事情，以确定方法调用不会做意料之外的事情。

HeadFirst：但是如果客户不知道他所处理的对象是哪一种，在不检查类型的情况下，他们又如何知道应该调用什么呢？

组合：如果你稍微有一点创意，就可以将你的方法架构起来，好让默认实现能够做一些有意义的事情。比方说，如果你的客户调用了getChild()，对组合来说，这个方法是有意义的。如果你把叶节点想象成没有孩子的对象，这个方法对叶节点来说也是有意义的。

HeadFirst：噢……聪明。但是，我听说一些客户其实很担心这个问题，所以他们对不同的对象用了不同的接口，这样就不会产生没有意义的方法调用了。这还算是组合模式吗？

组合：是的，这是更安全版本的组合模式，但是这需要客户先检查每个对象的类型，然后才进行方法的调用。

HeadFirst：请告诉我们更多的关于组合和叶节点对象的结构的事吧。

组合：通常是用树形结构，也就是一种层次结构。

根就是顶层的组合，然后往下是它的孩子，最末端是叶节点。

HeadFirst：孩子会不会反向指向它的父亲？

组合：是的，组件可以有一个指向父亲的指针，以便在游走时更容易。而且，如果引用某个孩子，你想从树形结构中删除这个孩子，你会需要父亲去删除它。一旦孩子有了指向父亲的引用，这做起来就很容易。

HeadFirst：在你的实现上，还真的有很多的事情需要考虑呢。在实现组合模式的时候，还有其他的问题吗？

组合：老实说，还有……其中之一就是孩子的次序。万一你有一个需要保持特定孩子次序的组合对象，就需要使用更复杂的管理方案来进行孩子的增加和删除，而且当你在这个层次结构内游走时，应该要更加小心。

HeadFirst：很好的观点，我根本没想到过。

组合：你想到过缓存（caching）吗？

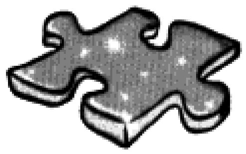
HeadFirst：缓存？

组合：是的，缓存。有时候，如果这个组合结构很复杂，或者遍历的代价太高，那么实现组合节点的缓存就很有帮助。比方说，如果你要不断地遍历一个组合，而且它的每一个子节点都需要进行某些计算，那你就应该使用缓存来临时保存结果，省去遍历的开支。

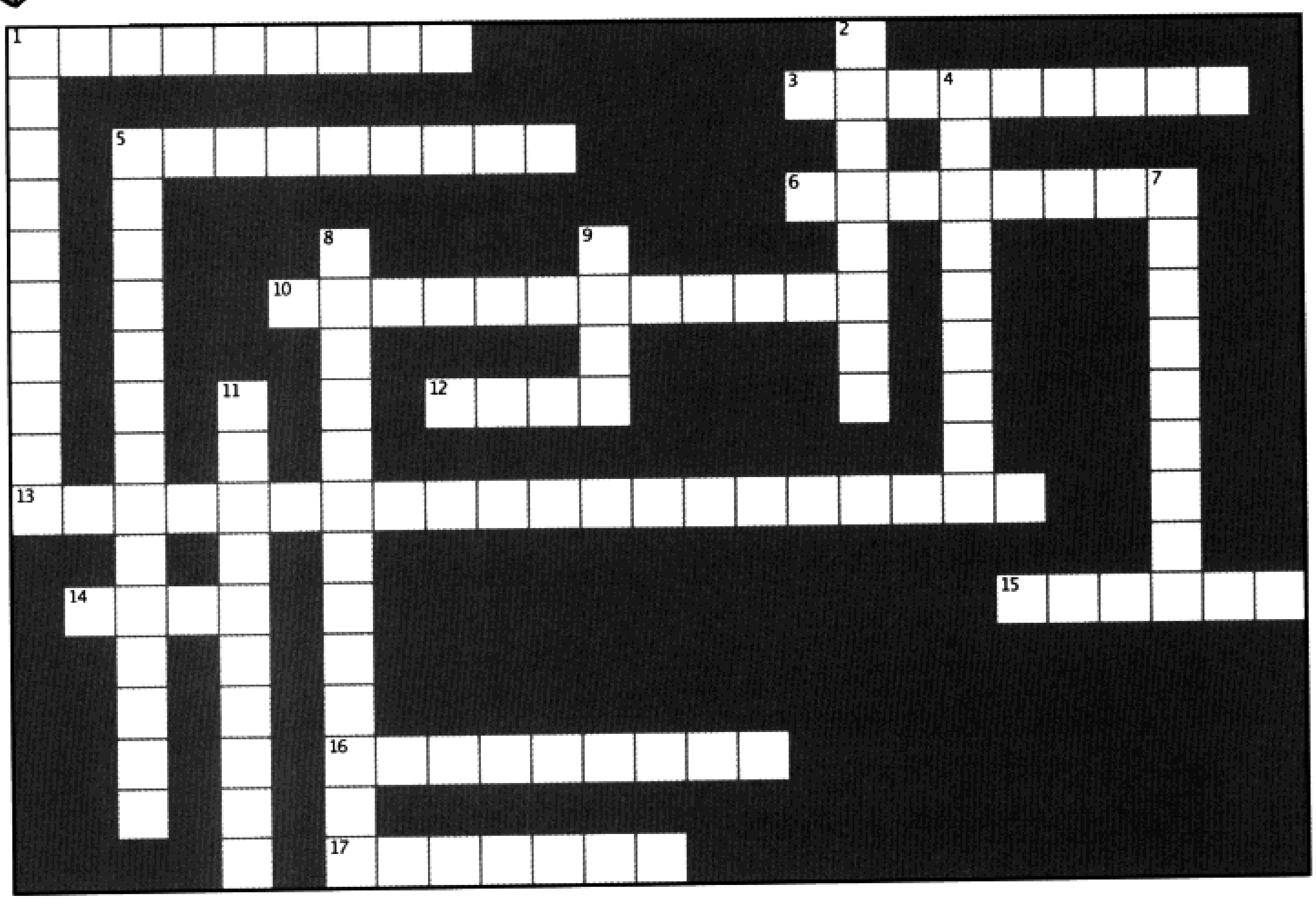
HeadFirst：哇！组合模式真的具有相当的内涵，远远超出我之前的想象。在我们结束之前，我还有最后一个问题：你认为你的最大强项是什么？

组合：我认为我让客户生活得更加简单。我的客户不再需要操心面对的是组合对象还是叶节点对象了，所以就不需要写一大堆if语句来保证他们对正确的对象调用了正确的方法。通常，他们只需要对整个结构调用一个方法并执行操作就可以了。

HeadFirst：听起来像是一个很重要的好处。毫无疑问，你是一个很有用的模式，可以帮助我们收集和管理对象。时间已经到了……非常感谢您的参与，别忘了继续关注其他的模式告白。



又到了休闲的时刻……



横排提示：

1. User interface packages often use this pattern for their components.
3. Collection and Iterator are in this package
5. We encapsulated this.
6. A separate object that can traverse a collection.
10. Merged with the Diner.
12. Has no children.
13. Name of principle that states only one responsibility per class.
14. Third company acquired.
15. A class should have only one reason to do this.
16. This class indirectly supports Iterator.
17. This menu caused us to change our entire implementation.

竖排提示：

1. A composite holds this.
2. We java-enabled her.
4. We deleted PancakeHouseMenuIterator because this class already provides an iterator.
5. The Iterator Pattern decouples the client from the aggregates __________.
7. CompositeIterator used a lot of this.
8. Iterators are usually created using this pattern.
9. A component can be a composite or this.
11. Hashtable and ArrayList both implement this interface.

连连看

请将下列模式和描述配对：

模式	描述
策略	客户可以将对象的集合以及个别的对象一视同仁
适配器	提供一个方式来遍历集合，而无须暴露集合的实现
迭代器	简化一群类的接口
外观	改变一个或多个类的接口
组合	当某个状态改变时，允许一群对象能被通知到
观察者	封装可互换的行为，并使用委托决定使用哪一个



设计箱内的工具

多了两个模式，两种很棒的方法来处理集合对象。

OO原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

为交互对象之间的松耦合设计而努力

类应该对扩展开放，对修改关闭。

依赖抽象，不要依赖具体类。

只和朋友交谈

别找我，我会找你

类应该只有一个改变的理由

又有一个新的重要原则，涉及到设计的改变。

OO模式

迭代器模式——提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

组合模式——允许你将对象组成树形结构来表现“整体/部分”的层次结构。组合能让客户以一致的方式处理个别对象和对象组合。

本章又同时介绍了两个模式。

要点

- 迭代器允许访问聚合的元素，而不需要暴露它的内部结构。
- 迭代器将遍历聚合的工作封装进一个对象中。
- 当使用迭代器的时候，我们依赖聚合提供遍历。
- 迭代器提供了一个通用的接口，让我们遍历聚合的项，当我们编码使用聚合的项时，就可以使用多态机制。
- 我们应该努力让一个类只分配一个责任。
- 组合模式提供一个结构，可同时包容个别对象和组合对象。
- 组合模式允许客户对个别对象以及组合对象一视同仁。
- 组合结构内的任意对象称为组件，组件可以是组合，也可以是叶节点。
- 在实现组合模式时，有许多设计上的折衷。你要根据需要平衡透明性和安全性。

习题解答

Sharpen your pencil

根据我们的printMenu()实现，下列哪一项为真？

☑ A. 我们是针对PancakeHouseMenu和DinerMenu的具体实现编码，而不是针对接口。

☐ B. 女招待没有实现Java女招待API，所以她没有遵守标准。

☑ C. 如果我们决定从DinerMenu切换到另一种菜单，此菜单的项是用Hashtable来存放的，我们会因此需要修改女招待中的许多代码。

☑ D. 女招待需要知道每个菜单如何表达内部的菜单项集合，这违反了封装。

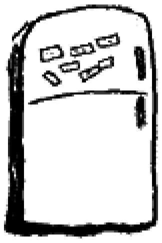
☑ E. 我们有重复的代码：printMenu()方法需要两个循环，来遍历两种不同的菜单。如果我们加上第三种菜单，我们就需要第三个循环。

☐ F. 这个实现并没有基于MXML（Menu XML），所以就没有办法互操作。

Sharpen your pencil

在看下一页之前，请很快写下为了能让这份代码符合我们的框架，我们要对它做的三件事情：

1. 实现Menu接口
2. 丢弃getItems()
3. 加上createIterator()，返回一个Iterator，以便遍历Hashtable的值。



代码帖解答

组合出“另一种”DinerMenu的迭代器

```
import java.util.Iterator;
import java.util.Calendar;

public class AlternatingDinerMenuIterator  implements Iterator  {

    MenuItem[] items;
    int position;

    public AlternatingDinerMenuIterator(MenuItem[] items)  {

        this.items = items;
        Calendar rightNow = Calendar.getInstance();
        position = rightNow.get(calendar.DAY_OF_WEEK) % 2;
    }

    public boolean hasNext() {

        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }

    public Object next() {

        MenuItem menuItem = items[position];
        position = position + 2;
        return menuItem;
    }

    public void remove() {

        throw new UnsupportedOperationException(
            "Alternating Diner Menu Iterator does not support remove()");
    }
}
```

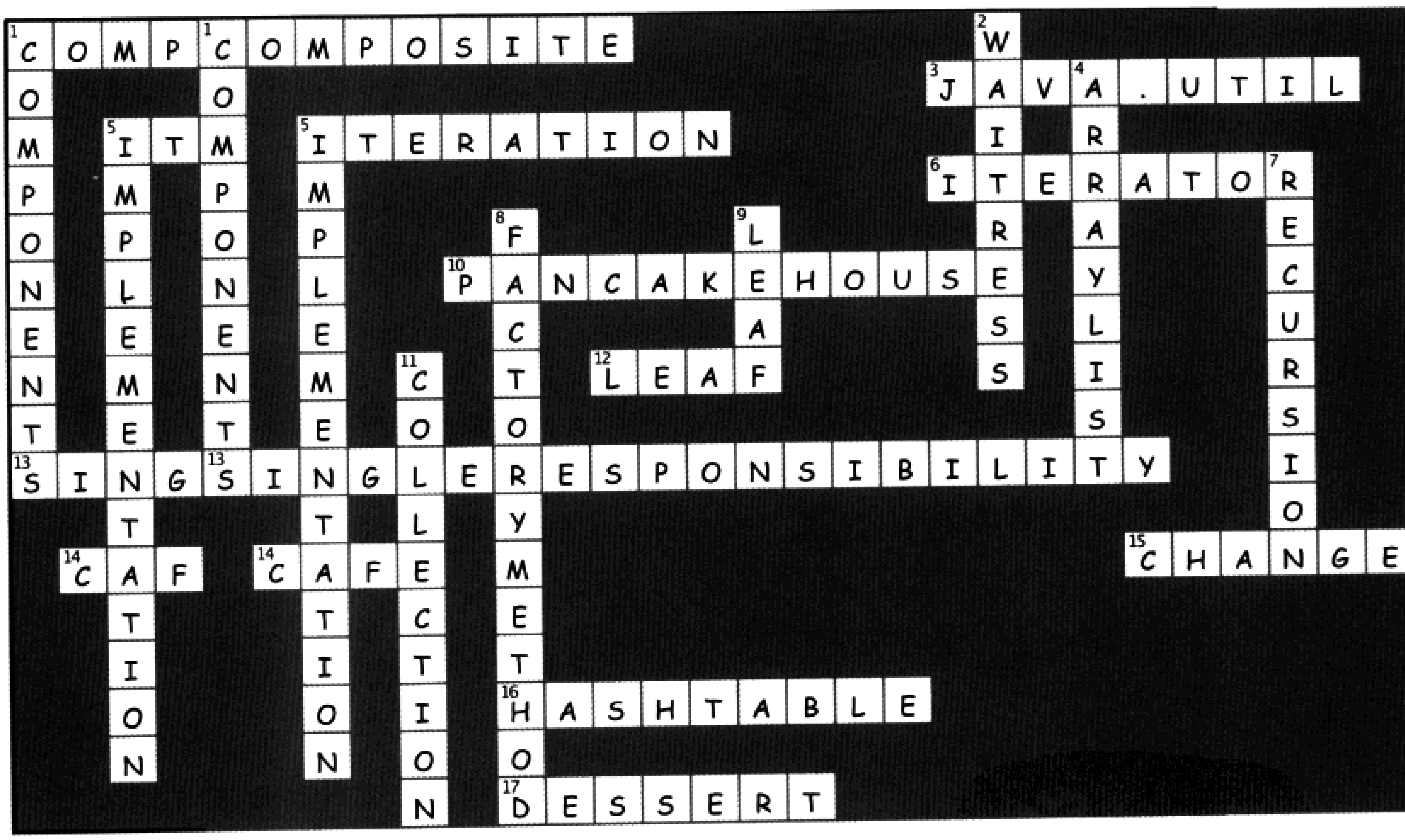
请注意，此迭代器实现不支持remove()

连连看

请将下列模式和描述配对：

模式	描述
策略	客户可以将对象的集合以及个别的对象一视同仁
适配器	提供一个方式来遍历集合，而无须暴露集合的实现
迭代器	简化一群类的接口
外观	改变一个或多个类的接口
组合	当某个状态改变时，允许一群对象能被通知到
观察者	封装可互换的行为，并使用委托决定使用哪一个

习题解答

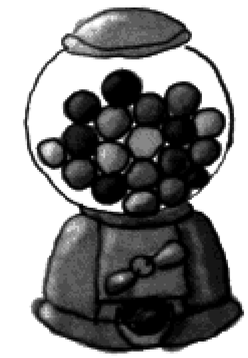

COMPOSITE
WAITRESS
JAVA.UTIL
ARRAYLIST
ITERATOR
RECURSION
ITERATION
IMPLEMENTATION
COMPONENT
FACTORY METHOD
LEAF
PANCAKEHOUSE
COLLECTION
SINGLE RESPONSIBILITY
CAFE
CHANGE
HASHTABLE
DESSERT

10 状态模式

事物的状态



基本常识：策略模式和状态模式是双胞胎，在出生时才分开。 你已经知道了，策略模式是围绕可以互换的算法来创建成功业务的。然而，状态走的是更崇高的路，它通过改变对象内部的状态来帮助对象控制自己的行为。它常常告诉它的对象客户“跟着我念：我很棒，我很聪明，我最优秀了……”

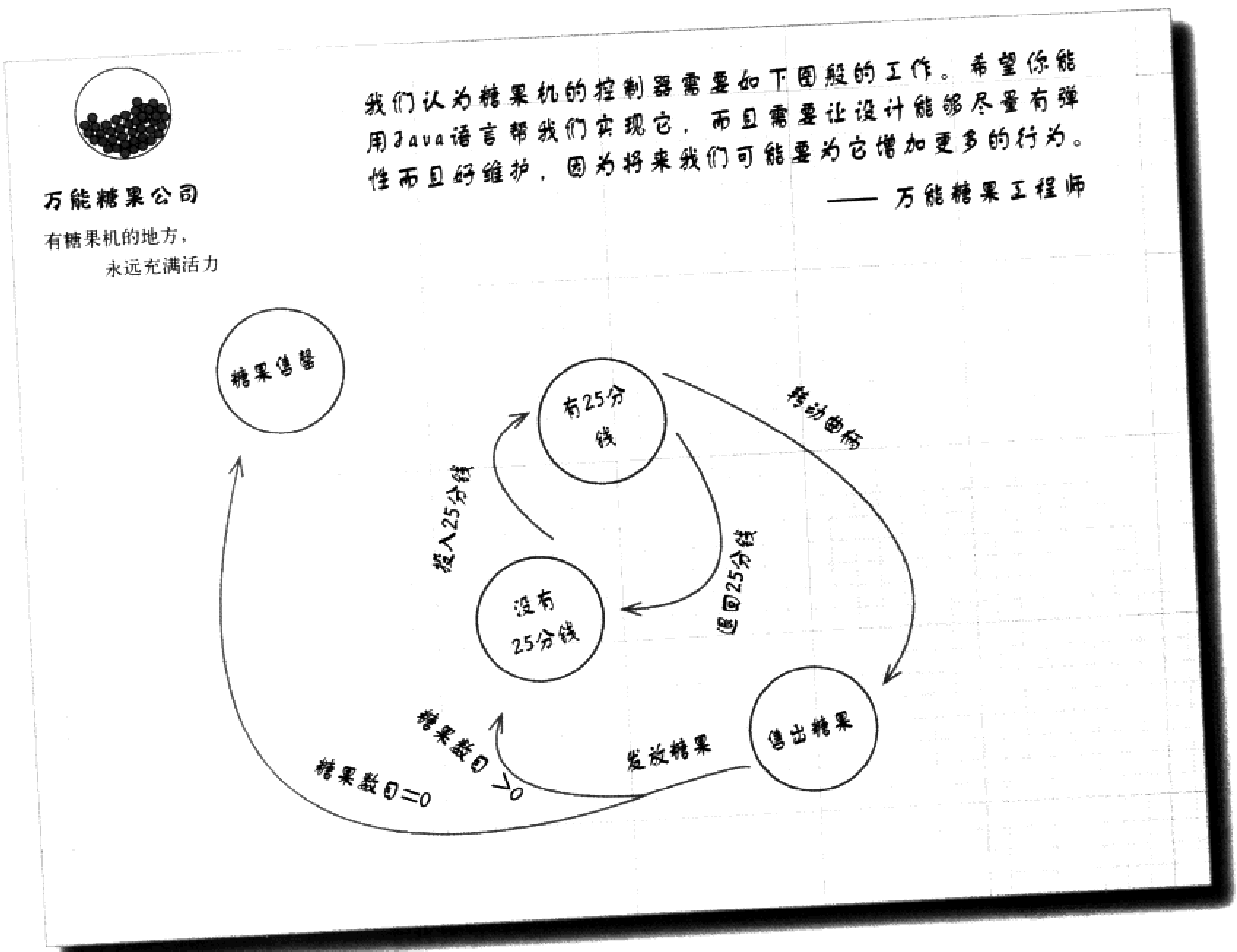


Java粉碎机

Java烤面包机已经落伍了，现在人们已经把Java创建在像糖果机这样真正的装置中。没错，糖果机已经进入了高科技时代。糖果机的主要制造厂商发现，只要把CPU放进机器中，就可以增加销售量、通过网络监测库存,并且能精准地得知客户的满意度。

至少这是他们单方面的说法——我们认为他们其实只是厌倦了公元1800年左右的科技，想要找些事情让他们的工作变得更有趣。

但是这些制造商都是糖果机的专家，并非软件专家，他们需要你的帮助：



办公室隔间对话



Anne：这张图像是一张状态图。

Joe：没错，每个圆圈都是一个状态.

Anne：……而每个箭头都是状态的转换。

Frank：慢点，你们两个。我已经好久没有接触状态图，忘得一干二净了！你们能提醒我状态图是干什么的吗？

Anne：当然可以了，Frank。你看到的圆圈，就是状态。“没有25分钱”大概就是糖果机的开始状态，等着你把钱放进来。每一个状态都代表机器不同的配置以某种方式行动，需要某些动作将目前的状态转换到另一个状态。

Joe：没错。看，要进入另一个状态，必须做某些事情，例如将25分钱的硬币放进机器中。所以你看到有一个箭头从“没有25分钱”指向“有25分钱”。

Frank：是的……

Joe：这就表示如果糖果机在“没有25分钱”的状态下，放进25分钱的硬币，就会进入“有25分钱”的状态。这就是状态的转换。

Frank：噢！我懂了！如果我是在“有25分钱”的状态，就可以转动曲柄改变到“售出糖果”状态，或者退还硬币回到“没有25分钱”状态。

Anne：就是这样！

Frank：这个状态图看起来并不太难。很明显我们有四个状态，而我认为我们也有四个动作，分别为：“投入25分钱”、“退回25分钱”、“转动曲柄”和“发放糖果”。但是……当我们发放的时候，要在“售出糖果”的状态中测试，是否糖果数目已经为零，来决定是否要进入到“糖果售罄”状态，或是进入“没有25分钱”状态。所以实际上，我们有五个状态转换。

Anne：测试糖果数目是否为零，也意味着我们必须持续地追踪糖果的数目。任何时候只要机器给出一颗糖果，都有可能是最后一颗糖果，如果是的话，我们就需要转换到“糖果售罄”状态。

Joe：也请不要忘了可以做没有意义的事，例如，当糖果机在“没有25分钱”状态的时候，试着去退回25分钱，或者是在糖果机内同时放进两个25分钱。

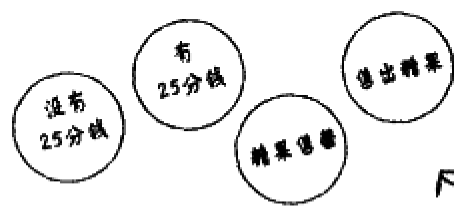
Frank：噢！这我倒没想到；我们也要注意到这部分。

Joe：对于任何一个可能的动作，我们都要检查，看看我们所处的状态和动作是否合适。这没问题！让我们开始将状态图映射成代码……

状态机101

我们如何从状态图得到真正的代码呢？下面是一个实现状态机（state machine）的简单介绍。

❶ 首先，找出所有的状态：



这就是状态——总共有四个。

❷ 接下来，创建一个实例变量来持有目前的状态，然后定义每个状态的值：

将“糖果售罄”简称为“售罄”（Sold out）。

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

每一个状态都用一个不同的整数代表。

……而这是一个实例变量，持有当前的状态。我们将它设置为“糖果售罄”状态，因为糖果机一开始拆箱并安装的时候，是没有装糖果的。

❸ 现在，我们将所有系统中可以发生的动作整合起来：

投入25分钱　　转动曲柄

退回25分钱

发放糖果

这些动作是糖果机的接口——这是你能对糖果机做的事情。

看看这个图，调用任何一个动作都会造成状态的转换。

发放糖果更多是糖果机的内部动作，机器自己调用自己。

❹ 现在，我们创建了一个类，它的作用就像是一个状态机。对每一个动作，我们都创建了一个对应的方法，这些方法利用条件语句来决定在每个状态内什么行为是恰当的。比如对“投入25分钱”这个动作来说，我们可以把对应方法写成下面的样子：

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another quarter");

    } else if (state == SOLD_OUT) {

        System.out.println("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

        System.out.println("Please wait, we're already giving you a gumball");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    }
}
```

每一个可能的状态都需要用条件语句检查……

……然后对每一个可能的状态展现适当的行为……

……但是也可以转换到另一个状态像状态图中所描绘的那样。

我们在这里所谈论的是一个通用的技巧：如何对对象内的状态建模——通过创建一个实例变量来持有状态值，并在方法内书写条件代码来处理不同状态。



在这一段简洁的说明之后，让我们开始实现糖果机吧！

写下代码

现在我们来实现糖果机。我们知道要利用实例变量持有当前的状态，然后需要处理所有可能发生的动作、行为和状态的转换。我们需要实现的动作包括：投入25分钱、退回25分钱、转动曲柄和发放糖果；也要检查糖果是否售罄。

这就是那四个状态。它们符合万能糖果公司的状态图。

这个实例变量跟踪当前状态，一开始被设置为“糖果售罄”。

我们还有第二个实例变量，用来追踪机器内的糖果数目。

构造器需要初始糖果库存量当做参数。如果库存量不为零的话，机器就会进入“没有25分钱”的状态，也就是说它等着别人投入25分钱。如果糖果数目为0的话，机器就会保持在“糖果售罄”的状态。

```java
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
```

现在我们开始将这些动作实现成方法……

当有25分钱投进来，就会执行这里……

如果已投入过25分钱，我们就告诉顾客；

如果是在“没有25分钱”的状态下，我们就接受25分钱，并将状态转换到“有25分钱”的状态。

```java
    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

如果顾客刚刚才买了糖果，就需要稍等一下，好让状态转换完毕，恢复到“没有25分钱”的状态。

如果糖果已经售罄，我们就拒绝收钱。

现在，如果顾客试着退回25分钱……

```
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
```

如果有25分钱，就把钱退出来，回到“没有25分钱”的状态。

如果没有25分钱的话，当然不能退出25分钱。

如果顾客已经转动曲柄，就不能再退钱了，他已经拿到糖果了！

如果糖果售罄，就不可能接受25分钱，当然也不可能退钱。

顾客试着转动曲柄……

```
public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}
```

别想骗过机器拿两次糖果。

我们需要先投入25分钱。

我们不能给糖果——已经没有任何糖果了。

成功！他们拿到糖果了。改变状态到“售出糖果”，然后调用机器的dispense()方法。

调用此方法，发放糖果。

```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}

// 这里是像toString()和refill()的其他的方法
}
```

我们正在“售出糖果”状态，给他们糖果！

我们在这里处理“糖果售罄”的情况。如果这是最后一颗糖果，我们就将机器的状态设置到“糖果售罄”；否则，就回到“没有25分钱”状态。

这些都不应该发生，但如果顾客这么做了，他们得到的是错误消息，而不是得到糖果。

内部测试

感觉它像是使用思虑周密的方法学构造的牢不可破的设计，你不觉得吗？在我们将它交给万能糖果公司，安装到实际的糖果机器内之前，让我们先做一个小小的内部测试。测试程序是这样的：

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

总共装了5颗糖果。

打印出机器的状态。

投入一枚25分钱硬币……

转动曲柄，我们应该拿到糖果。

再一次打印出机器的状态。

投入一枚25分钱硬币……

要求机器退钱。

转动曲柄，我们应该拿不到糖果。

再一次打印出机器的状态。

投入一枚25分钱硬币……

转动曲柄，我们应该拿到糖果。

投入一枚25分钱硬币……

转动曲柄，我们应该拿到糖果。

要求机器退钱。

再一次打印出机器的状态。

放进两枚25分钱硬币……

转动曲柄，我们应该拿到糖果。

现在做压力测试……

再一次打印出机器的状态。

File Edit Window Help mightygumball.com

```
%java GumballMachineTestDrive

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

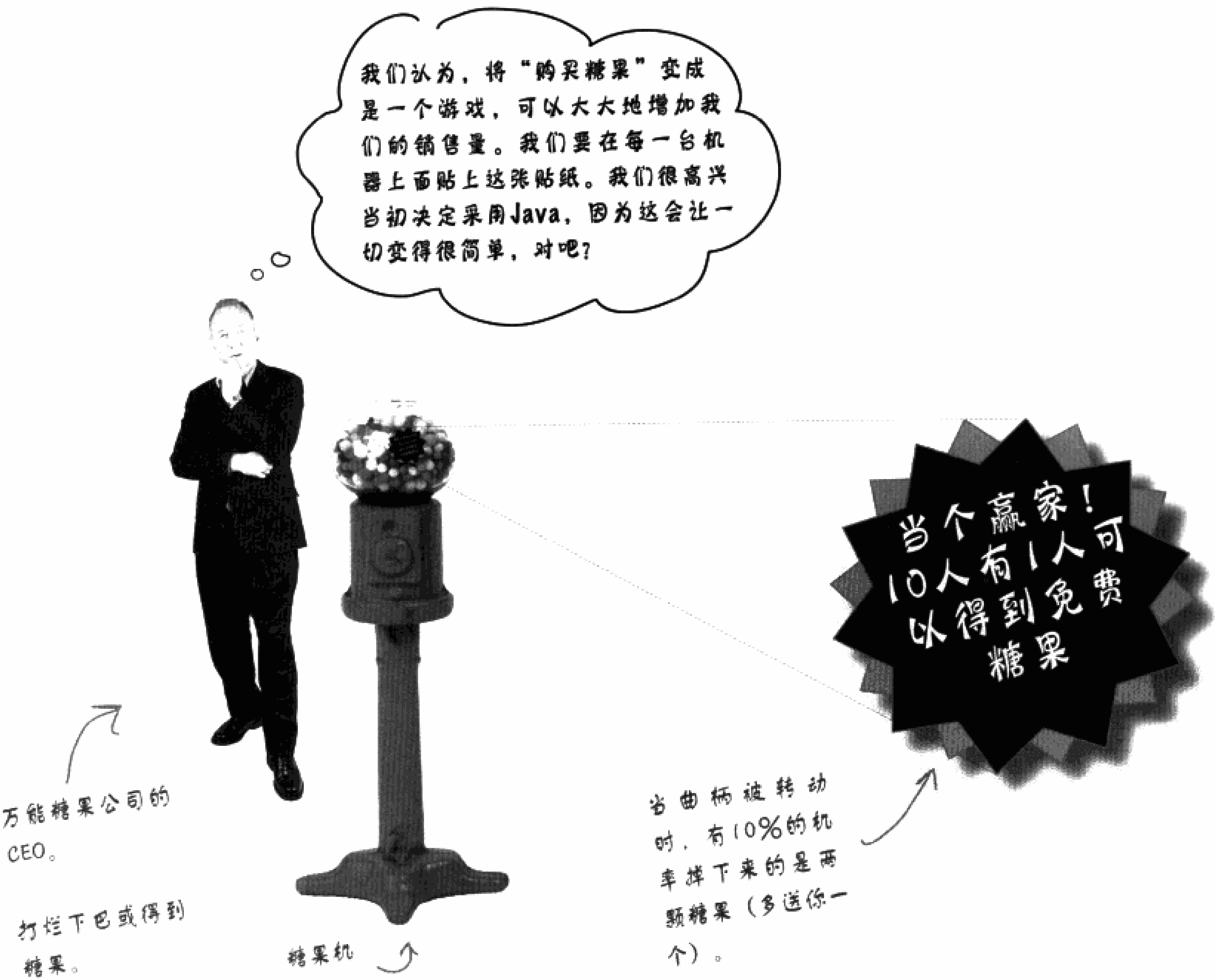
You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
```

该来的躲不掉……变更请求！

万能糖果公司已经将你的代码放进他们的新机器中，然后让他们的质保专家进行测试。到目前为止，在他们看来一切都很顺利。

事实上，实在是太顺利了，所以他们想要变点花样……



设计谜题

为万能糖果公司的机器绘制一个状态图，处理这个十次赢一次的竞赛。在这个竞赛中，“售出糖果”状态有10%的机率会导致掉下两颗糖果，而不是一颗。在你继续下一步之前，请将你的答案和我们的解答做对比（在本章的最后），以确定我们的看法一致……

万能糖果公司

有糖果机的地方，

永远充满活力

使用万能糖果公司的文具来画你的状态图。

混乱的状态……

使用一种考虑周详的方法学写糖果机的代码，并不意味着这份代码就容易扩展。事实上，当你回顾这些代码，并开始考虑要如何修改它时……

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;
```

首先，你必须加上一个新的状态，称为"赢家"。这还不算太麻烦……

```
public void insertQuarter() {
    // 这里加入投币代码
}

public void ejectQuarter() {
    // 这里加入退币代码
}

public void turnCrank() {
    // 这里加入转动曲柄代码
}

public void dispense() {
    // 这里加入发放糖果代码
}
```

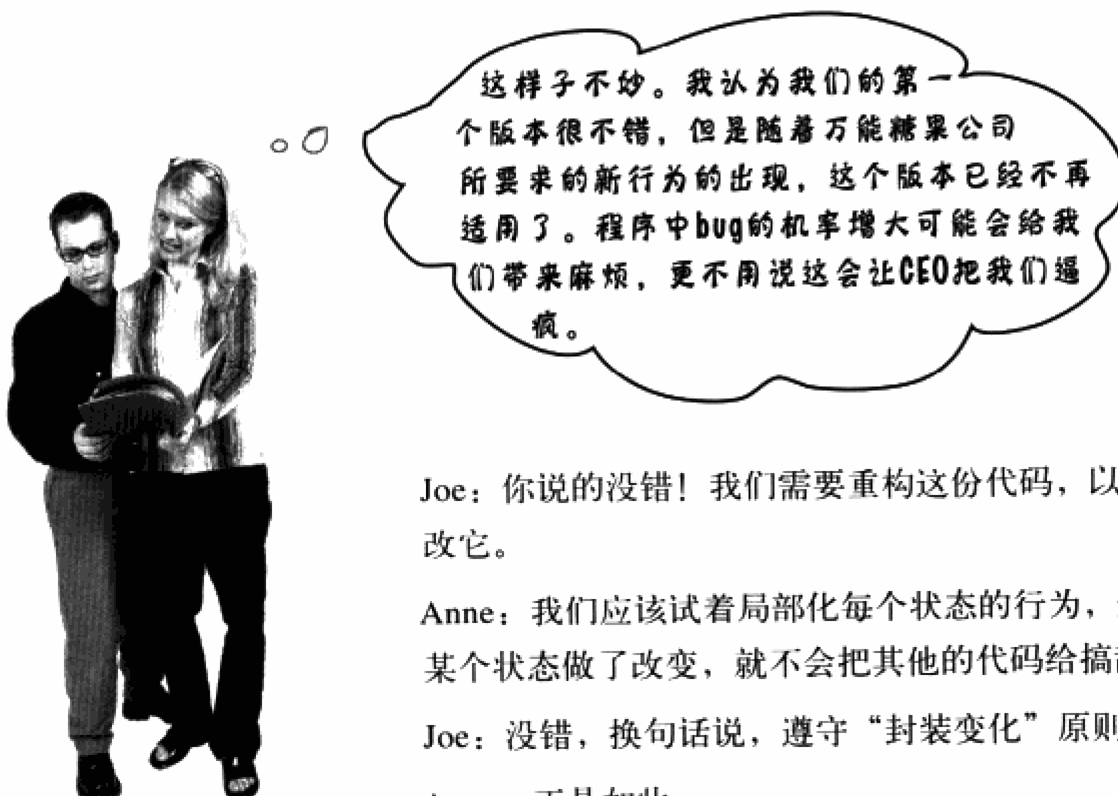
然后呢，你必须在每个方法中加入一个新的条件判断来处理"赢家"状态；这可有你忙的了。

turnCrack()尤其会变得一团乱，因为你必须加上代码来检查目前的顾客是否是赢家，然后再决定是切换到赢家状态还是售出糖果状态。

Sharpen your pencil

下列哪一项描述了我们实现的状态？（多选）

- ❑ A. 这份代码确实没有遵守开放-关闭原则。
- ❑ B. 这份代码会让Fortran 程序员感到骄傲。
- ❑ C. 这个设计其实不符合面向对象。
- ❑ D. 状态转换被埋藏在条件语句中，所以并不明显。
- ❑ E. 我们还没有把会改变的那部分包装来。
- ❑ F. 未来加入的代码很有可能会导致bug。



Joe：你说的没错！我们需要重构这份代码，以便我们能容易地维护和修改它。

Anne：我们应该试着局部化每个状态的行为，这样一来，如果我们针对某个状态做了改变，就不会把其他的代码给搞乱了。

Joe：没错，换句话说，遵守"封装变化"原则。

Anne：正是如此。

Joe：如果我们将每个状态的行为都放在各自的类中，那么每个状态只要实现它自己的动作就可以了。

Anne：对。或许糖果机只需要委托给代表当前状态的状态对象。

Joe：哇！你真行：这不正是"多用组合，少用继承"吗？我们应用了更多的原则。

Anne：呵呵！我并没有百分之百确定就要这么做，但是我想我们已经有正确的方向了。

Joe：我正在想这是否可以使添加新状态更容易呢？

Anne：我认为可以……我们还是需要改变代码，但是改变将局限在小范围内。因为加入一个新的状态，就意味着我们要加入一个新的类还有可能要改变一些转换。

Joe：听起来不错。让我们动手进行新的设计吧！

新的设计

我们的计划是这样的：不要维护我们现有的代码，我们重写它以便于将状态对象封装在各自的类中，然后在动作发生时委托给当前状态。

我们在这里遵照我们的设计原则，所以最后应该得到一个容易维护的设计。我们要做的事情是：

1. **首先，我们定义一个State接口。在这个接口内，糖果机的每个动作都有一个对应的方法。**

2. **然后为机器中的每个状态实现状态类。这些类将负责在对应的状态下进行机器的行为。**

3. **最后，我们要摆脱旧的条件代码，取而代之的方式是，将动作委托到状态类。**

你将会看到，我们不仅遵守了设计原则，实际上我们还实现了状态模式。在重新完成代码之后我们再来了解状态模式的正式定义……

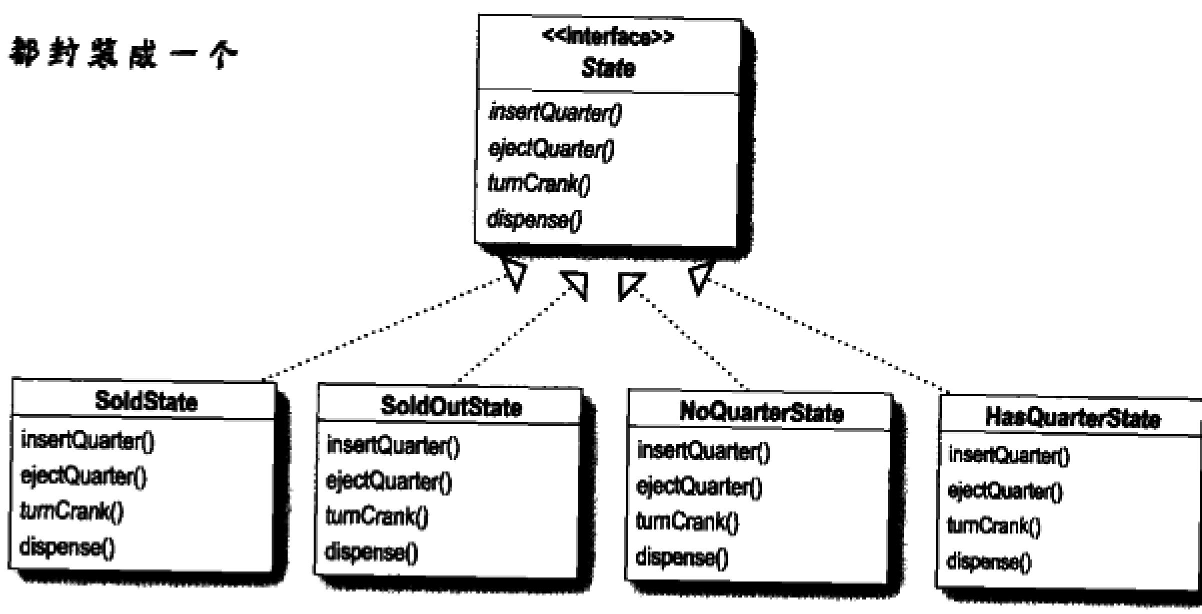


定义状态接口和类

首先，让我们创建一个State接口，所有的状态都必须实现这个接口：

这就是所有状态的接口。这些方法直接映射到糖果机上可能发生的动作（这些方法和之前代码里的一样）。

然后将设计中的每个状态都封装成一个类，每个都实现State接口。



想要理清我们需要什么状态，可以参考一下之前写的代码……

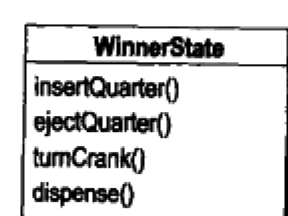
```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

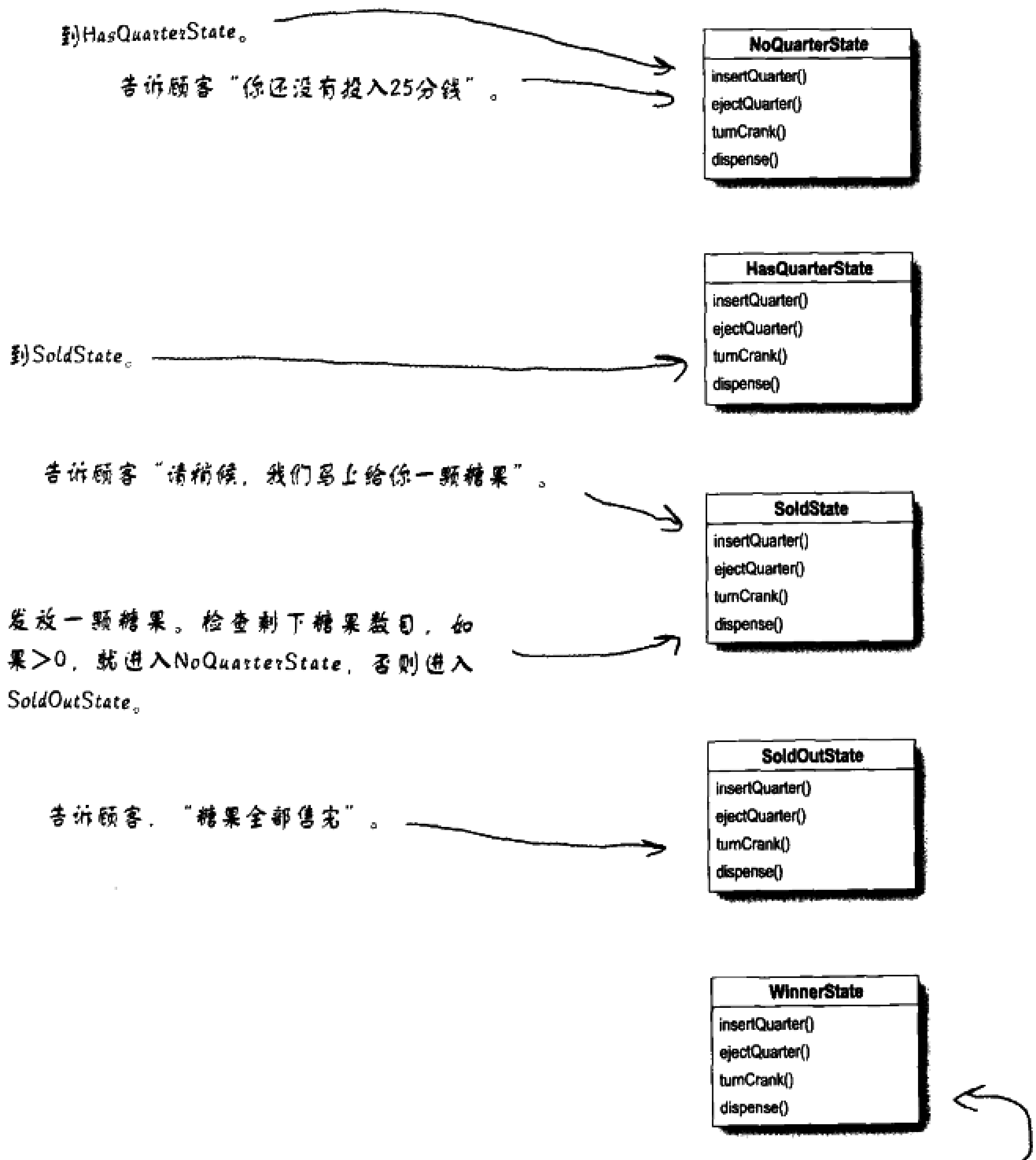
……然后我们将每个状态直接映射到一个类。

别忘了，我们也需要一个新的“赢家”状态（当然它也要实现State接口）。在我们完成第一个版本的糖果机的重新实现之后，再回来处理这部分。



Sharpen your pencil

想要实现我们的状态，我们首先需要指定当每一个动作被调用时，类的行为是哪一个。请在下面这张图上，为每个类的每个动作的行为加上注释。我们已经先帮你填写了其中的几个。



你可以继续把这张图填完，我们稍后也会继续实现这张图。

实现我们的状态类

现在是实现一个状态的时候了：我们知道我们要的行为是什么，我们只需要把它变成代码。我们打算完全遵守所写下的状态机代码，但是这一次是分散在不同的类中。

让我们从NoQuarterState开始：

首先我们需要实现State接口。

我们通过构造器得到糖果机的引用，然后将它记录在实例变量中。

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

如果有人投入了25分钱，我们就打印出一条消息，说我们接受了25分钱，然后改变机器的状态到HasQuarterState。

你马上就会看到这是如何工作的。

如果没给钱，就不能要求退钱。

如果没给钱，就不能要求糖果。

如果没得到钱，我们就不能发放糖果。



重新改造糖果机

在完成这些状态类之前，我们要重新改造糖果机——好让你了解这一切的原理。我们从状态相关的实例变量开始动手，然后把原来使用整数代表的状态改为状态对象：

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

在这个糖果机中，我们更新代码以使用新的类，而不再使用静态整数。除了一个类持有整数，而另一个是对象……之外，两者的代码其实很类似。

旧代码

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

}
```

新代码

所有的状态对象都是在构造器中创建并赋值的。

这个实例变量现在持有一个状态对象，而不是一个整数。

完整的糖果机类……

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    // 这里有更多的方法，其中包括每一个状态的getter……
}
```

所有的状态都在这里……

……以及实例变量state。

这个count实例变量记录机器内装有多少糖果——开始机器是没有装糖果的。

构造器取得糖果的初始数目并把它存放在一个实例变量中。

每一种状态也都创建一个状态实例。

如果超过0颗糖果，我们就把状态设为NoQuarterState。

现在这些动作变得很容易实现了。我们只是委托到当前状态。

请注意，我们不需要在GumballMachine中准备一个dispense()的动作方法，因为这只是一个内部的动作；用户不可以直接要求机器发放糖果。但我们是在状态对象的turnCrank()方法中调用dispense()方法的。

这个方法允许其他的对象（像我们的状态对象）将机器的状态转换到不同的状态。

这个机器提供了一个releaseBall()的辅助方法来释放出糖果，并将count实例变量的值减1。

这包括了像getNoQuarterState()这样用来取得每个对象的状态的方法；还包括可以取得糖果的数目的getCount()方法。

实现更多的状态

现在你应该开始对糖果机和状态之间是如何配合的有点儿感觉了。让我们实现HasQuarterState（有25分钱）和SoldState（售出糖果）类……

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

当状态被实例化的时候，我们要传入GumballMachine的引用来作为参数。

这是一个对此状态不恰当的动作。

退出顾客的25分钱，并将状态转换到NoQuarterState状态。

当曲柄被转动时，我们就调用它的setState()方法，并传入SoldState对象作为参数，将机器的状态转换到SoldState状态。这个SoldState对象可以通过getSoldState()方法取得（每个状态都有一个getter方法）。

这是此状态的另一个不恰当的动作。

现在，让我们来看看SoldState类……

对此状态来说，这些都是不恰当的动作。

```
public class SoldState implements State {
    // 构造器和实例变量在这里

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

真正的工作在这里。

我们现在是在SoldState状态，也就是说顾客已经付钱了。所以我们首先需要机器发放糖果。

我们问机器糖果的剩余数目是多少，然后将状态转换到NoQuarterState或者SoldOutState。

BRAIN POWER

让我们来回头看看糖果机的实现。如果曲柄被转动了，但是没有成功（比方说顾客没有先投入25分钱的硬币）。在这种情况下，尽管没有必要，但我们还是会调用dispense()方法。对于这个问题你要如何修改呢？

Sharpen your pencil

我们还剩下一个没有实现的类：SoldOutState（糖果售罄状态）。你何不来实现它呢？小心地弄清楚糖果机在每种情况下应该如何反应。在继续下一页之前，请先检查一下你的答案……

```
public class SoldOutState implements [          ] {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {



    }

    public void insertQuarter() {



    }

    public void ejectQuarter() {



    }

    public void turnCrank() {



    }

    public void dispense() {



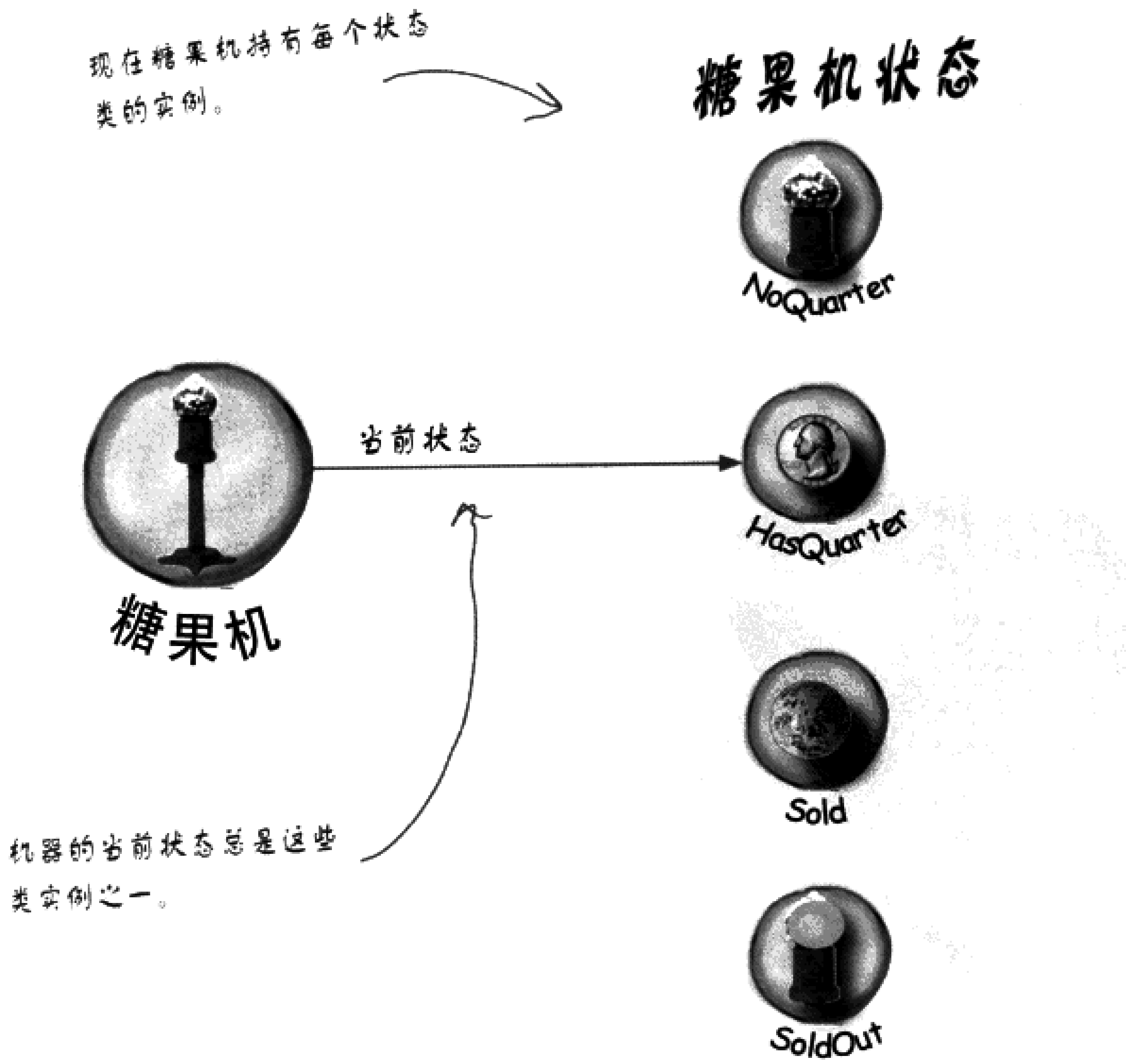
    }
}
```

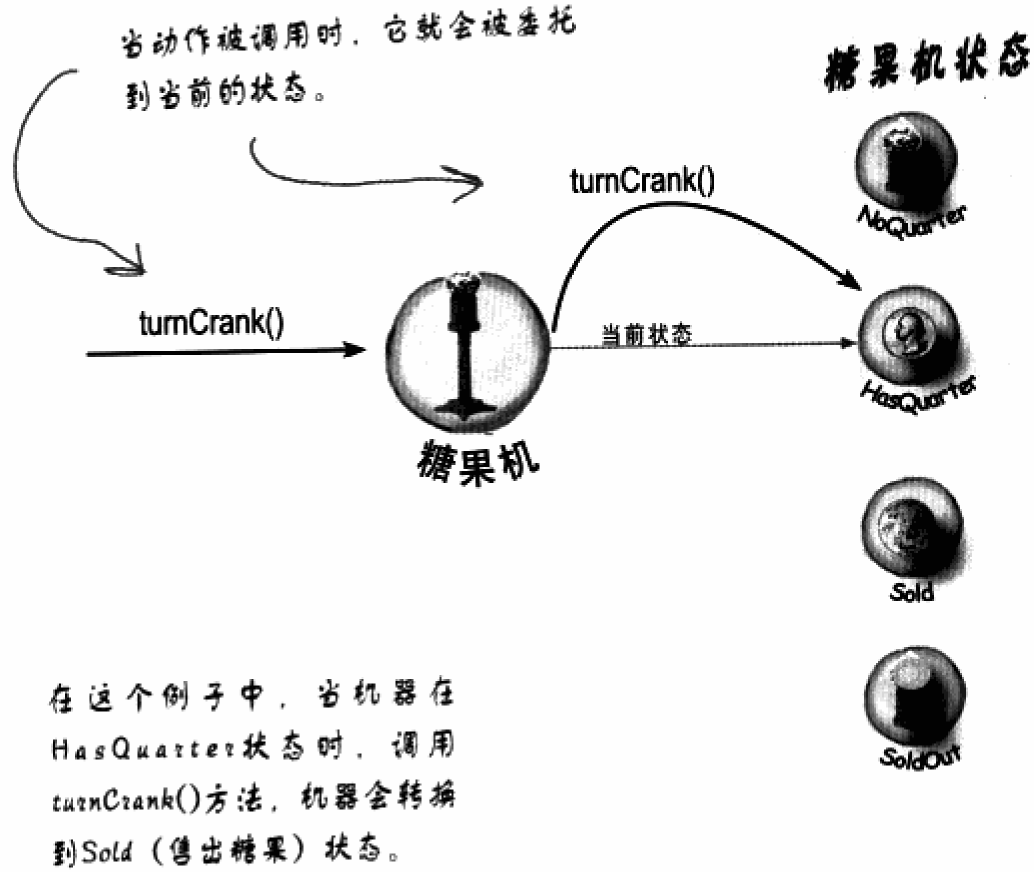
检查一下，到目前为止我们已经做了哪些事情……

你现在有了一个糖果机的实现，它在结构上和前一个版本差异颇大，但是功能上却是一样的。通过从结构上改变实现，你已经做到了以下几点。

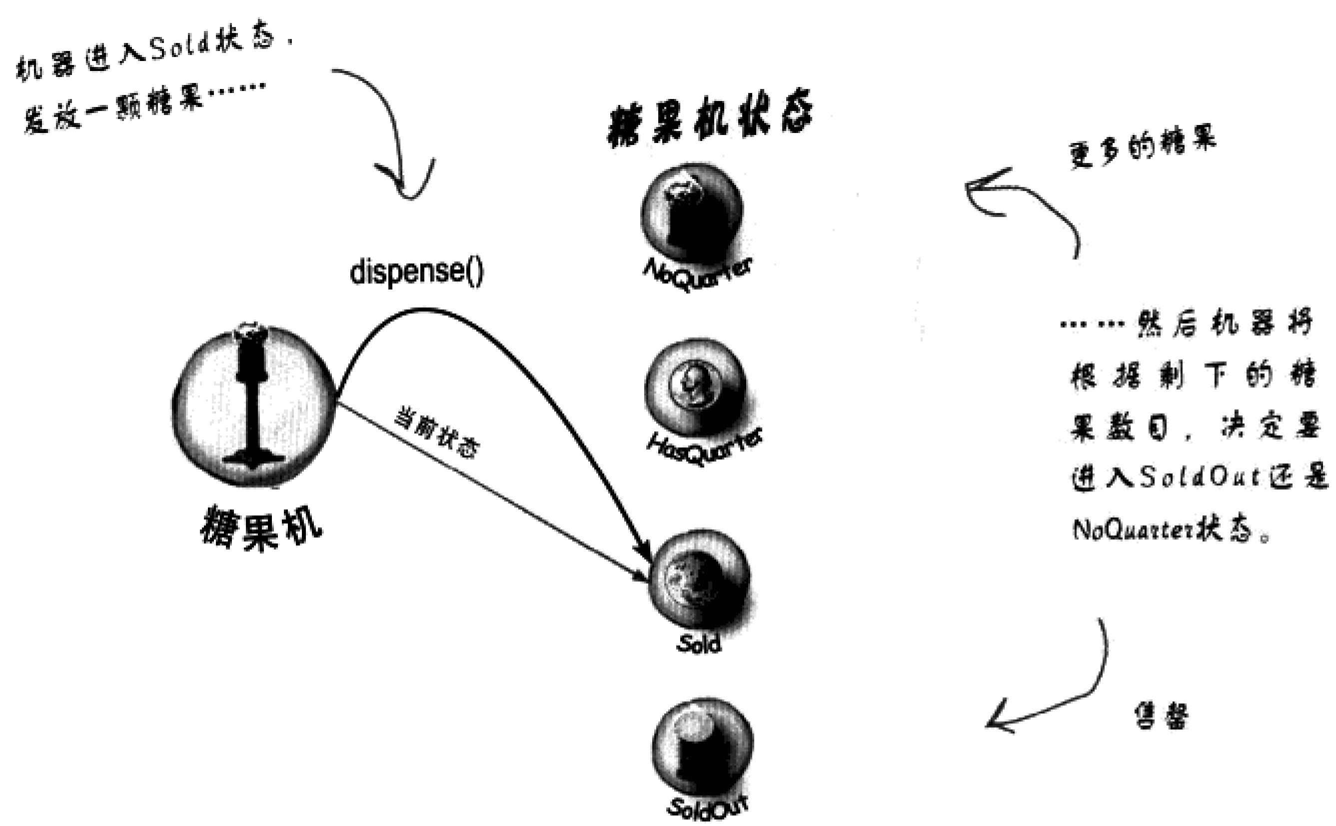
- 将每个状态的行为局部化到它自己的类中。
- 将容易产生问题的if语句删除，以方便日后的维护。
- 让每一个状态“对修改关闭”，让糖果机“对扩展开放”，因为可以加入新的状态类（我们马上就这么做）。
- 创建一个新的代码基和类结构，这更能映射万能糖果公司的图，而且更容易阅读和理解。

现在，再多检查一些我们所做的功能面：




当动作被调用时，它就会被委托到当前的状态。
糖果机状态
turnCrank()
turnCrank()
当前状态
糖果机
NoQuarter
HasQuarter
Sold
SoldOut
在这个例子中，当机器在HasQuarter状态时，调用turnCrank()方法，机器会转换到Sold（售出糖果）状态。

转换到Sold状态

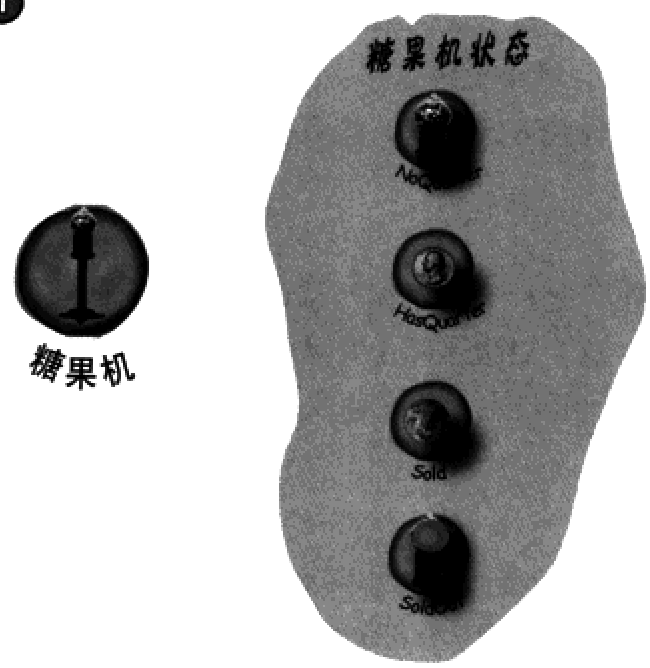

机器进入Sold状态，发放一颗糖果……
糖果机状态
dispense()
当前状态
糖果机
NoQuarter
HasQuarter
Sold
SoldOut
更多的糖果
……然后机器将根据剩下的糖果数目，决定要进入SoldOut还是NoQuarter状态。
售罄

Sharpen your pencil

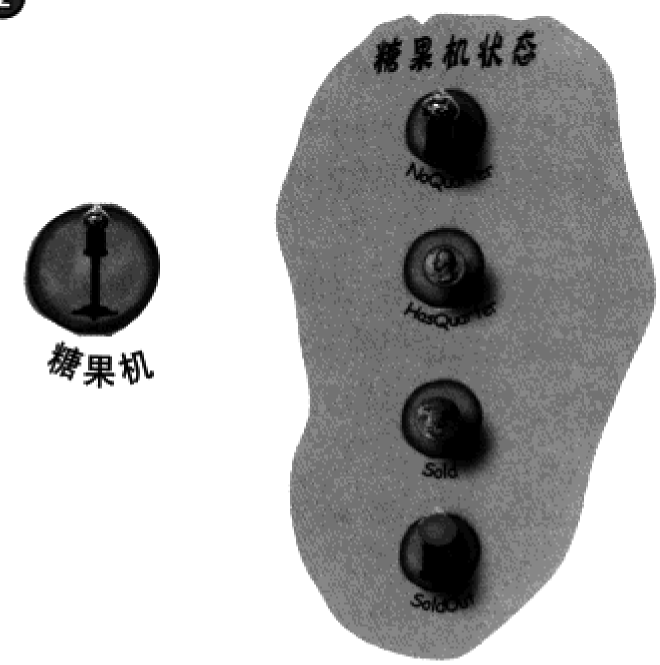
幕后花絮：自我导览

从NoQuarter状态开始追踪糖果机的工作步骤。也请利用机器的动作和输出为图加上说明。在这个练习中，你可以假设机器中有很多糖果。

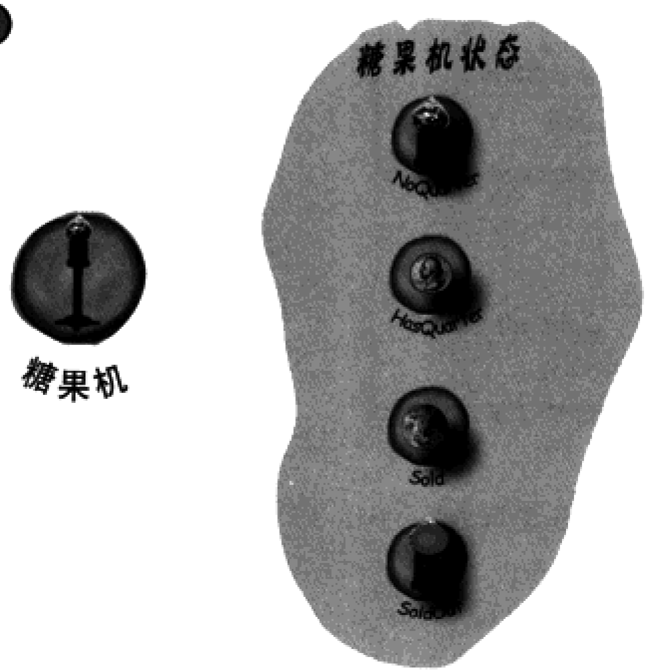
1



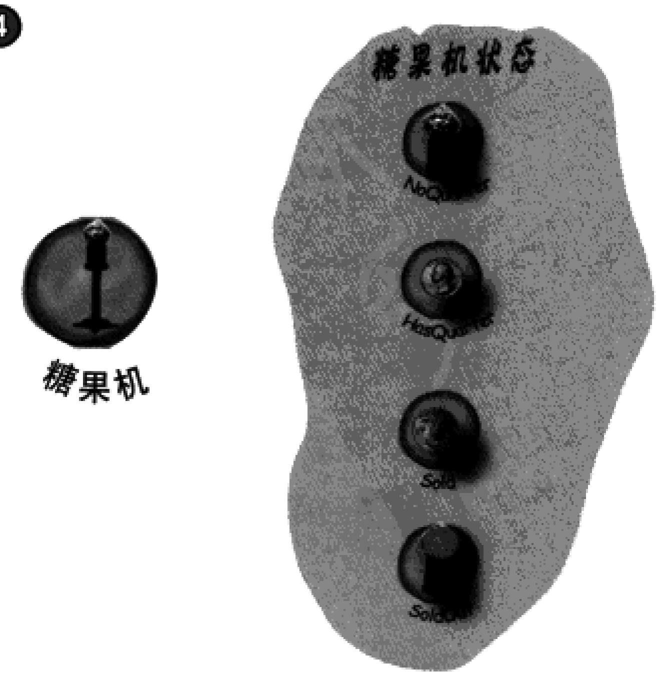
2



3



4



定义状态模式

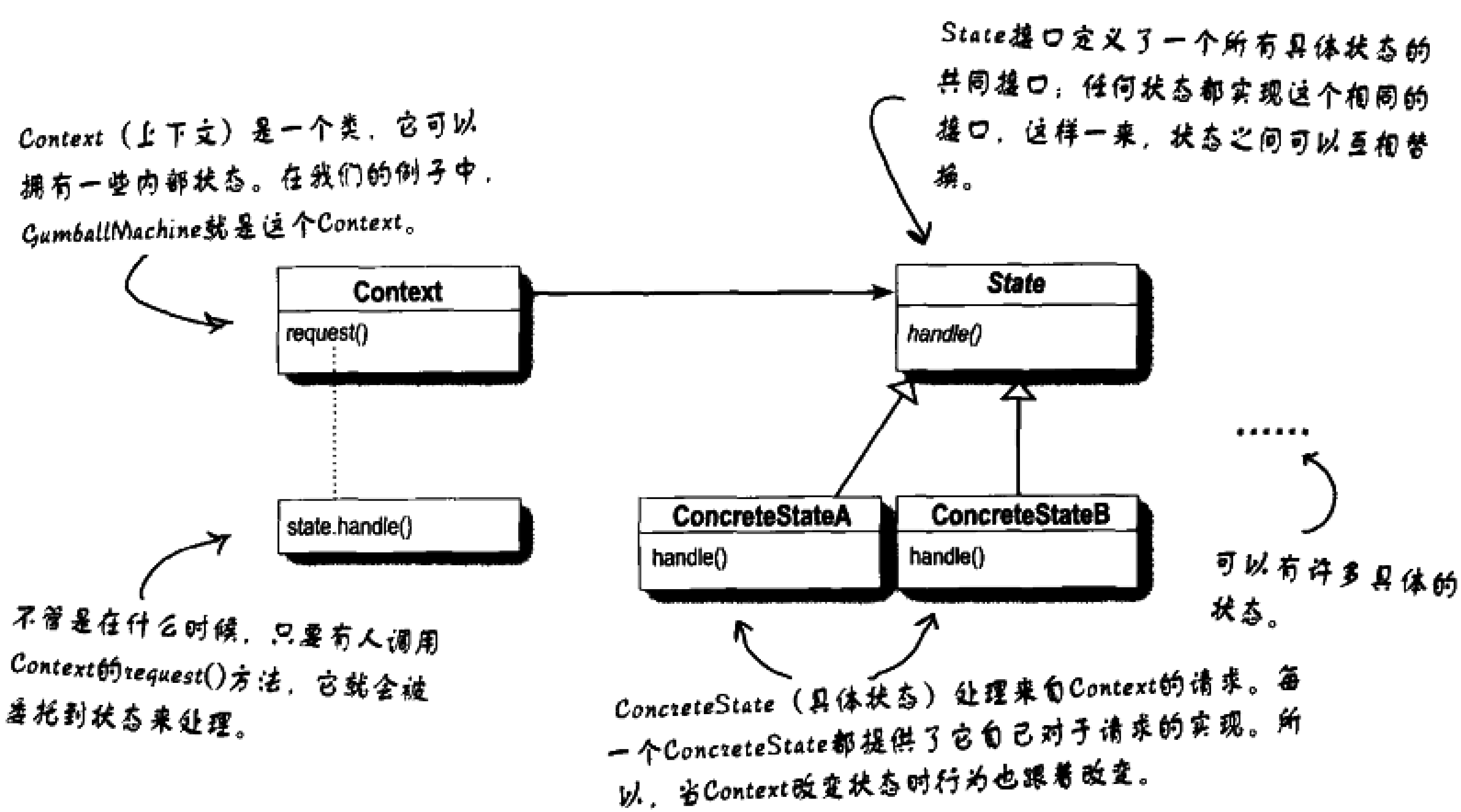
是的，这是真的，我们刚刚实现了状态模式！现在，让我们来看看它是怎么一回事：

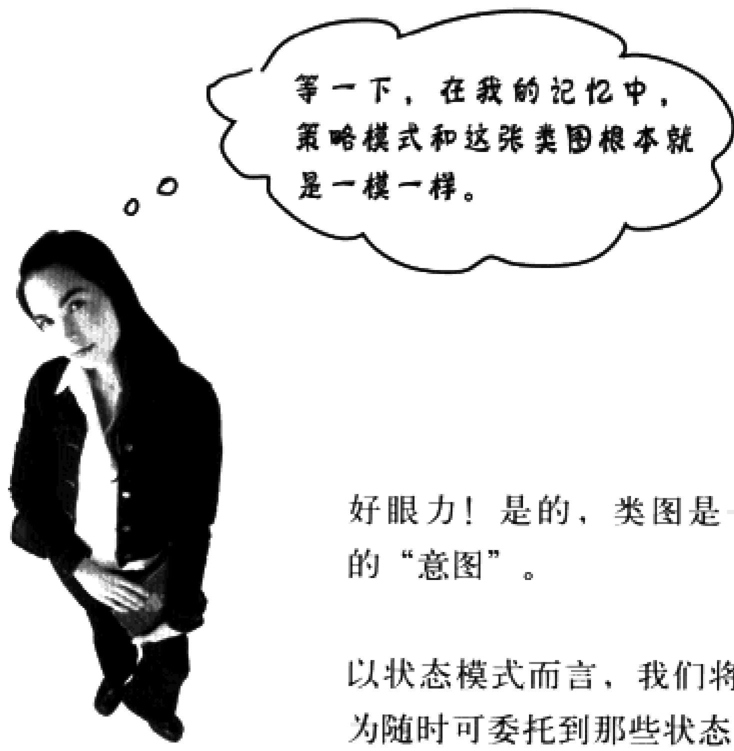
> **状态模式**允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

这个描述中的第一部分附有相当多的涵义，是吧？因为这个模式将状态封装成为独立的类，并将动作委托到代表当前状态的对象，我们知道行为会随着内部状态而改变。糖果机提供了一个很好的例子：当糖果机是在NoQuarterState或HasQuarterState两种不同的状态时，你投入25分钱，就会得到不同的行为（机器接受25分钱和机器拒绝25分钱）。

而这个定义中的第二部分呢？一个对象“看起来好像修改了它的类”是什么意思呢？从客户的视角来看：如果说你使用的对象能够完全改变它的行为，那么你会觉得，这个对象实际上是从别的类实例化而来的。然而，实际上，你知道我们是在使用组合通过简单引用不同的状态对象来造成类改变的假象。

好了，现在就让我们检查状态模式的类图：





好眼力！是的，类图是一样的，但是这两个模式的差别在于它们的“意图”。

以状态模式而言，我们将一群行为封装在状态对象中，context的行为随时可委托到那些状态对象中的一个。随着时间的流逝，当前状态在状态对象集合中游走改变，以反映出context内部的状态，因此，context的行为也会跟着改变。但是context的客户对于状态对象了解不多，甚至根本是浑然不觉。

而以策略模式而言，客户通常主动指定Context所要组合的策略对象是哪一个。现在，固然策略模式让我们具有弹性，能够在运行时改变策略，但对于某个context对象来说，通常都只有一个最适当的策略对象。比方说，在第1章，有些鸭子（例如绿头鸭）被设置成利用典型的飞翔行为进行飞翔，而有些鸭子（例如橡皮鸭和诱饵鸭）使用的飞翔行为只能让他们紧贴地面。

一般来说，我们把策略模式想成是除了继承之外的一种弹性替代方案。如果你使用继承定义了一个类的行为，你将被这个行为困住，甚至要修改它都很难。有了策略模式，你可以通过组合不同的对象来改变行为。

我们把状态模式想成是不用在context中放置许多条件判断的替代方案。通过将行为包装进状态对象中，你可以通过在context内简单地改变状态对象来改变context的行为。

there are no Dumb Questions

问：在GumballMachine中，状态决定了下一个状态应该是什么。ConcreteState总是决定接下来的状态是什么吗？

答：不，并非总是如此，Context也可以决定状态转换的流向。

一般来讲，当状态转换是固定的时候，就适合放在Context中；然而，当转换是更动态的时候，通常就会放在状态类中（例如，在GumballMachine中，由运行时糖果的数目来决定状态要转换到NoQuarter还是SoldOut）。

将状态转换放在状态类中的缺点是：状态类之间产生了依赖。在我们的GumballMachine实现中，我们试图通过使用Context上的getter方法把依赖减到最小，而不是显式硬编码具体状态类。

请注意，在做这个决策的同时，也等于是在为另一件事情做决策：当系统进化时，究竟哪个类是对修改封闭（Context还是状态类）的。

问：客户会直接和状态交互吗？

答：不会。状态是用在Context中来代表它的内部状态以及行为的，所以只有Context才会对状态提出请求。客户不会直接改变Context的状态。全盘了解状态是Context的工作，客户根本不了解，所以不会直接和状态联系。

问：如果在我的程序中Context有许多实例，这些实例之间可以共享状态对象吗？

答：是的，绝对可以，事实上这是很常见的做法。但唯一的前提是，你的状态对象不能持有它们自己的内部状态；否则就不能共享。

想要共享状态，你需要把每个状态都指定到静态的实例变量中。如果你的状态需要利用到Context中的方法或者实例变量，你还必须在每个handler()方法内传入一个context的引用。

问：使用状态模式似乎总是增加我们设计中类的数目。请看GumballMachine的例子，新版本比旧版本多出了许多类！

答：没错，在个别的状态类中封装状态行为，结果总是增加这个设计中类的数目。这就是为了要获取弹性而付出的代价。除非你的代码是一次性的，可以用完就扔掉（是呀！才怪！），那么其实状态模式的设计是绝对值得的。其实真正重要的是你暴露给客户的类数目，而且我们有办法将这些额外的状态类全都隐藏起来。

让我们看一下另一种做法：如果你有一个应用，它有很多状态，但是你决定不将这些状态封装在不同的对象中，那么你就会得到巨大的、整块的条件语句。这会让你的代码不容易维护和理解。通过使用许多对象，你可以让状态变得很干净，在以后理解和维护它们时，就可以省下很多的工夫。

问：状态模式类图显示State是一个抽象类，但你不是使用接口实现糖果机状态的吗？

答：是的。如果我们没有共同的功能可以放进抽象类中，就会使用接口。在你实现状态模式时，很可能想使用抽象类。这么一来，当你以后需要在抽象类中加入新的方法时就很容易，不需要打破具体状态的实现。

十次抽中一次的游戏，尚未解决……

别忘了，我们还没有完事呢。我们还有一个游戏在等待实现；然而，我们已经实现了状态模式，所以实现这个游戏应该易如反掌。首先，我们要在GumballMachine类中加入一个状态：

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;

    // 这里有一些方法
}
```

你需要在这里加进一个新的WinnerState状态，然后在构造器中将它初始化。

别忘了在这里提供一个WinnerState的getter方法。

现在让我们实现WinnerState类本身，其实它很像SoldState类：

```
public class WinnerState implements State {

    // 实例变量和构造器

    // insertQuarter错误信息

    // ejectQuarter错误信息

    // turnCrank错误信息

public void dispense() {
        System.out.println("YOU'RE A WINNER! You get two gumballs for your quarter");
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
```

就跟SoldState一样。

我们在这里释放出两颗糖果，然后进入NoQuarterState 或SoldOutState。

如果还有第二颗糖果的话，我们就把它释放出来。

完成这个游戏

我们还要再做一个改变：我们需要实现机会随机数，还要增加一个进入WinnerState状态的转换。这两件事情都要加进HasQuarterState，因为顾客会从这个状态中转动曲柄：

```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

首先我们增加一个随机数产生器，产生10%赢的机会……

然后决定这个顾客是否赢了。

如果赢了，而且有足够的糖果可以让他一次得到两颗的话，我们就进入WinnerState状态；否则，就进入SoldState状态（就跟平常一样）。

哇！实现起来真是容易！我们刚刚为GumballMachine增加了一个新的状态，并实现了这个新的状态。要做的事情只是实现我们的机会游戏，并转换到正确的状态。看来我们新的代码策略已经奏效了……

向万能糖果公司的CEO做展示

万能糖果公司的CEO来访，来看看我们的新糖果机代码的演示。希望这些状态都没问题！我们要让这个展示简短而甜蜜（CEO们的注意力可不会停留太久），但希望时间能够足够长，至少让我们赢一次！

这个代码其实没有改，我们只是把它缩短了一些。

再来一次，让糖果机一开始就装了5颗糖果。

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

我们希望能赢，所以一直投钱并转动曲柄。然后打印出糖果机的状态……

整个开发团队都在会议室外面屏息以待，想知道基于状态模式的设计是否能赢得青睐。






```
File Edit Window Help Whenisagumballajawbreaker?
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
%
```

这是老天爷的眷顾吗：在我们向CEO展示的时候，我们不是赢了一次，而是赢了两次！

there are no Dumb Questions

问：我们为什么需要WinnerState？为什么不直接在SoldState中发放两颗糖果？

答： 这是一个好问题。这两个状态几乎一样，唯一的差别在于，WinnerState状态会发放两颗糖果。你当然可以将发放两颗糖果的代码放在SoldState中，当然这么做有缺点，因为你等于是将两个状态用一个状态类来代表。这样做你牺牲了状态类的清晰易懂来减少一些冗余代码。你也应该考虑到在前面的章节中所学到的原则：一个类，一个责任。将WinnerState状态的责任放进SoldState状态中，你等于是让SoldState状态具有两个责任。那么促销方案结束之后或者赢家的机率改变之后，你又该怎么办呢？所以，这必须用你的智慧来做折衷。



精神检查……

是的，万能糖果公司的CEO或许需要去做精神检查，但这不是我们所要说的。在推出我们的黄金版本之前，让我们再检查看看GumballMachine还有哪些方面需要改进：

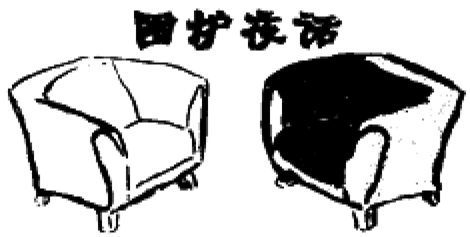
- 我们在售出糖果和赢家状态中，有许多重复的代码。我们必须把这部分清理一下。要怎么做呢？我们可以把State设计成抽象类，然后把方法的默认行为放在其中；毕竟，像是“你已经投入25分钱”这类的消息，不会被顾客看见。所以，所有的“错误响应”行为都可以写得具有通用性，并放在抽象的State类中供子类继承。

可恶！我是糖果机，不是电脑。你要搞清楚！Jim。

- dispense()方法即使是在没有25分钱时曲柄被转动的情况下也总是会被调用。我们可以轻易地修改这部分，做法是让turnCrank()返回一个布尔值，或者引入异常。你认为哪一种做法比较好？

- 状态转换的所有智能被放在状态类中，这可能导致什么问题？我们要将逻辑移进糖果机中吗？这有什么优缺点？

- 你会实例化许多的GumballMachine对象吗？如果是的话，你可能想要将状态的实例移到静态的实例变量中共享。这需要对GumballMachine和State做怎样的改变？



今夜话题：策略模式与状态模式重聚

策略　**状态**

策略： 老兄，你听说了我来自第1章吗？

状态： 是的，我听说了。

策略： 我刚去帮了模板方法一个忙——他们要我帮他们结束那个章节。言归正传，我的高贵的老兄，近来如何？

状态： 没什么变化——我还是在帮类的忙，让他们在不同的状态中展现不同的行为。

策略： 我不这么认为，你看起来就像是在抄袭我，只是换个词罢了。你想想：我允许对象能够通过组合和委托来拥有不同的行为或算法。你只是在抄袭我罢了。

状态： 我承认我们所做的事情绝对有关系，但是我的意图和你的完全不一样。我教客户使用组合和委托的做法是完全不一样的。

策略： 是吗？怎么说？我不了解。

状态： 如果你能别花那么多时间在自己身上，或许你就能了解我所说的。总而言之，想想看你是如何工作的：你有一个可以实例化的类，而且通常给它一个实现某些行为的策略对象。像是在第1章你处理呱呱叫的行为，对吗？真正的鸭子就拿到真正的呱呱叫行为，橡皮鸭子拿到吱吱叫的呱呱叫行为。

策略： 是的，那是很精细的活儿……我相信你一定能够看出来，为什么这比继承你的行为更有威力，你说是吧？

状态： 是的，当然了。现在，你来了解一下我的工作方式，它是截然不同的。

策略： 很抱歉，你需要解释一下你的工作。

策略 | **状态**

状态：好吧！当我的Context对象被创建之后，我可以告诉它们从什么状态开始，然后它们会随着时间而改变自己的状态。

策略：嘿！别这样，我也可以在运行时改变行为；毕竟这正是组合的目的！

状态：当然你也能这么做，但是我的做法是利用许多不同的状态对象；我的Context对象会随着时间而改变状态，而任何的状态改变都是定义好的。换句话说，“改变行为”这件事是建立在我的方案中的——这就是我的工作方式！

策略：好吧！我承认，我并没有鼓励我的对象拥有一组定义良好的状态转换。事实上，我通常会去控制我的对象使用什么策略。

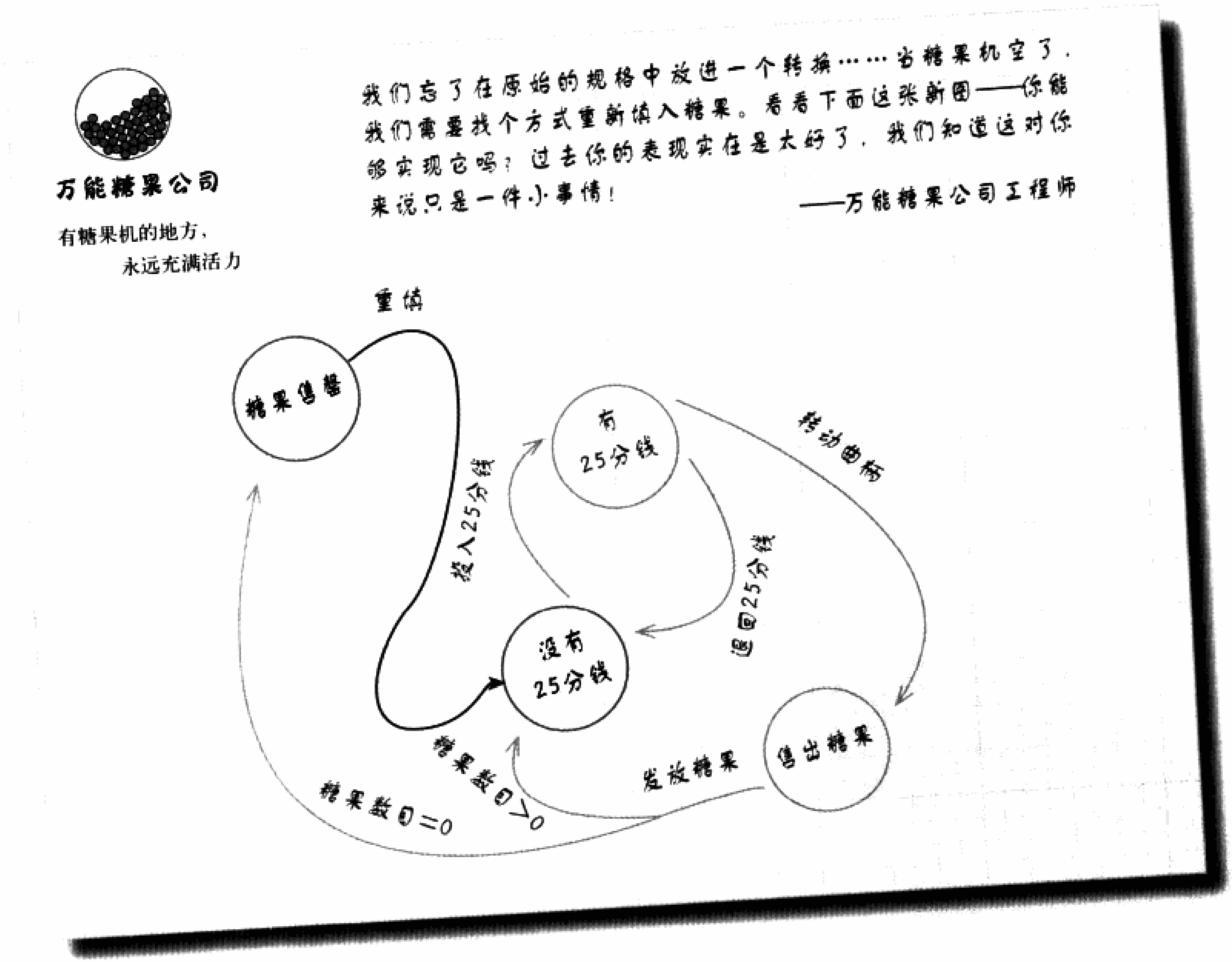
状态：看吧！我已经说过了我们在结构上很像，但是我们做事情的意图是十分不同的。面对这个事实吧，我们两个在这个世界上都有用处。

策略：是呀！继续做你的美梦吧，我的老兄。你好像以为自己和我一样是个大模式，但事实上，我可是在第1章就登场了；而你却是在第10章才有机会出现。我的意思是，有多少人能够真的把这本书看到第10章？

状态：开什么玩笑？这可是“HeadFirst”系列书籍，而这一系列的书都超棒。当然读者们会读到第10章！

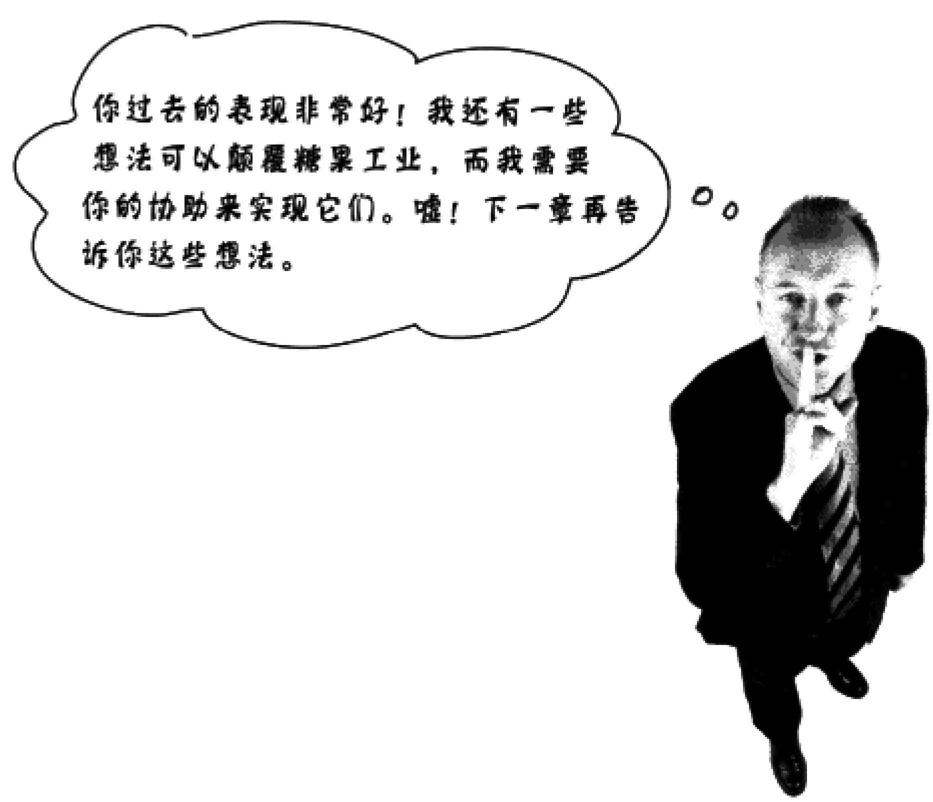
策略：这就是你，老兄，一直都在做梦。

我们差点儿忘了！


万能糖果公司
有糖果机的地方，
永远充满活力
我们忘了在原始的规格中放进一个转换……当糖果机空了，我们需要找个方式重新填入糖果。看看下面这张新图——你能够实现它吗？过去你的表现实在是太好了，我们知道这对你来说只是一件小事情！
——万能糖果公司工程师
重填
糖果售罄
投入25分钱
有25分钱
转动曲柄
退回25分钱
没有25分钱
售出糖果
发放糖果
糖果数目>0
糖果数目=0

Sharpen your pencil

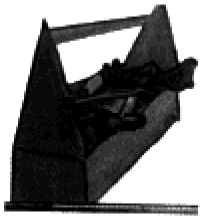
我们需要你为糖果机写一个重填糖果的refill()方法。这个方法需要一个变量——所要填入机器中的糖果数目。它应该能更新糖果机内的糖果数目，并重设机器的状态。



连连看

请将下列模式和描述配对：

模式	描述
状态	将可以互换的行为封装起来，然后使用委托的方法，决定使用哪一个行为
策略	由子类决定如何实现算法中的某些步骤
模板方法	封装基于状态的行为，并将行为委托到当前状态



设计箱内的工具

又到了另一个章节的结尾。你所懂的模式已经足够帮你轻松通过任何工作面试了！

OO基础

OO原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

为交互对象之间的松耦合设计而努力

类应该对扩展开放，对修改关闭。

依赖抽象，不要依赖具体类。

只和朋友交谈

别找我，我会找你

类应该只有一个改变的理由

本章没有新的原则，让你有时间回顾一下曾经学过的原则。

OO模式

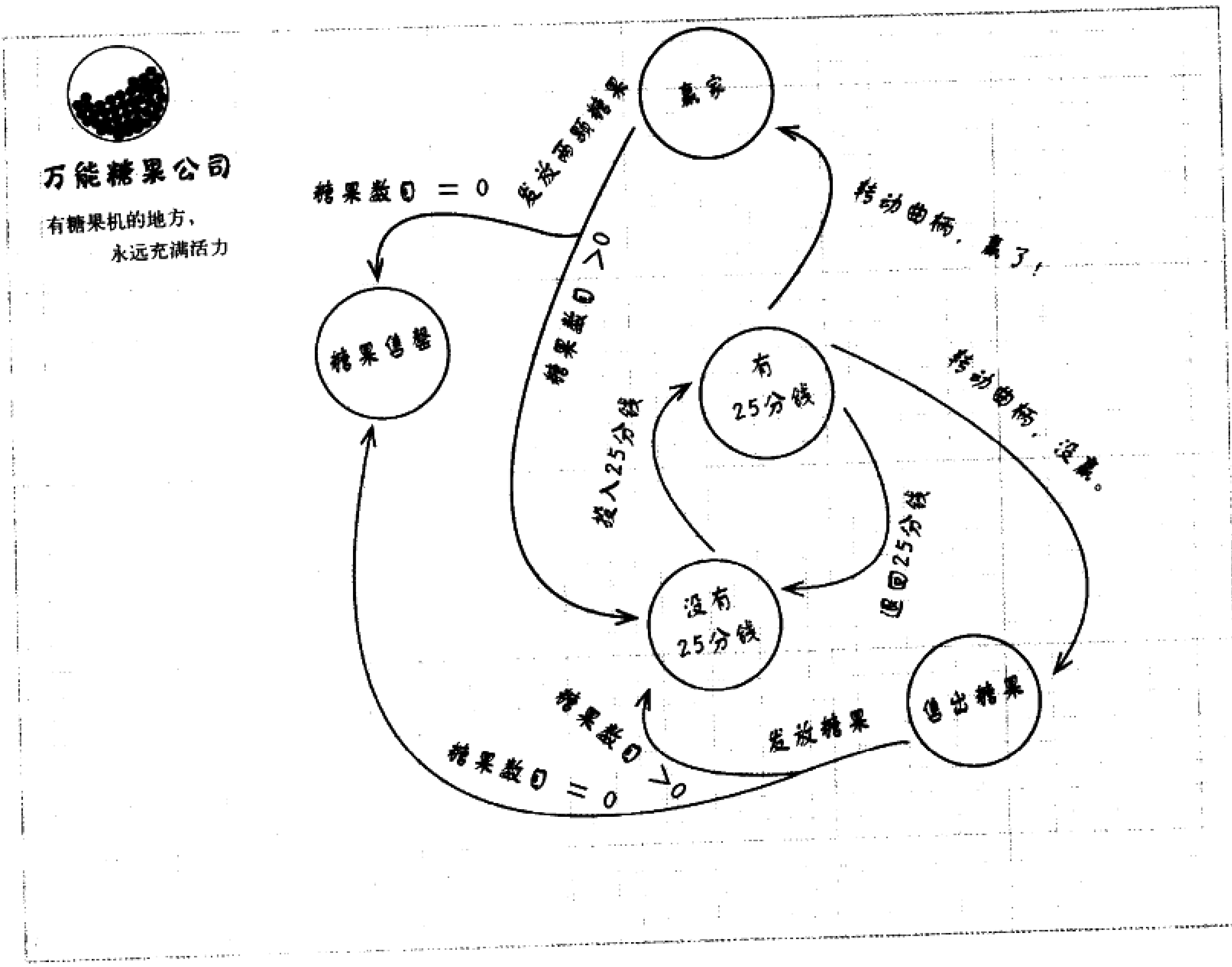
状态模式——允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

这是我们的新模式。如果你需要在一个类中管理状态，状态模式提供了封装状态的技巧。

要点

- 状态模式允许一个对象基于内部状态而拥有不同的行为。
- 和程序状态机（PSM）不同，状态模式用类代表状态。
- Context会将行为委托给当前状态对象。
- 通过将每个状态封装进一个类，我们把以后需要做的任何改变局部化了。
- 状态模式和策略模式有相同的类图，但是它们的意图不同。
- 策略模式通常会用行为或算法来配置Context类。
- 状态模式允许Context随着状态的改变而改变行为。
- 状态转换可以由State类或Context类控制。
- 使用状态模式通常会导致设计中类的数目大量增加。
- 状态类可以被多个Context实例共享。

习题解答


万能糖果公司
有糖果机的地方，
永远充满活力
赢家
发放两颗糖果
糖果数目 = 0
糖果数目 > 0
糖果售罄
转动曲柄，赢了！
有
25分钱
转动曲柄，没赢。
投入25分钱
退回25分钱
没有
25分钱
售出糖果
发放糖果
糖果数目 > 0
糖果数目 = 0

习题解答

Sharpen your pencil

下列哪一项描述了我们实现的状态？（多选）

- ☑ A. 这份代码确实没有遵守开放-关闭原则。
- ☑ B. 这份代码会让Fortran 程序员感到骄傲。
- ☑ C. 这个设计其实不符合面向对象。
- ☑ D. 状态转换被埋藏在条件语句中，所以并不明显。
- ☑ E. 我们还没有把会改变的那部分包装来。
- ☑ F. 未来加入的代码很有可能会导致bug。

Sharpen your pencil

我们还剩下一个没有实现的类：SoldOutState（糖果售罄状态）。你何不来实现它呢？小心地弄清楚糖果机在每种情况下应该有怎样的行为。在继续下一页之前，请先检查一下你的答案……

在糖果售罄状态下，除非有人重新填充糖果机，否则我不能做任何事情。

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

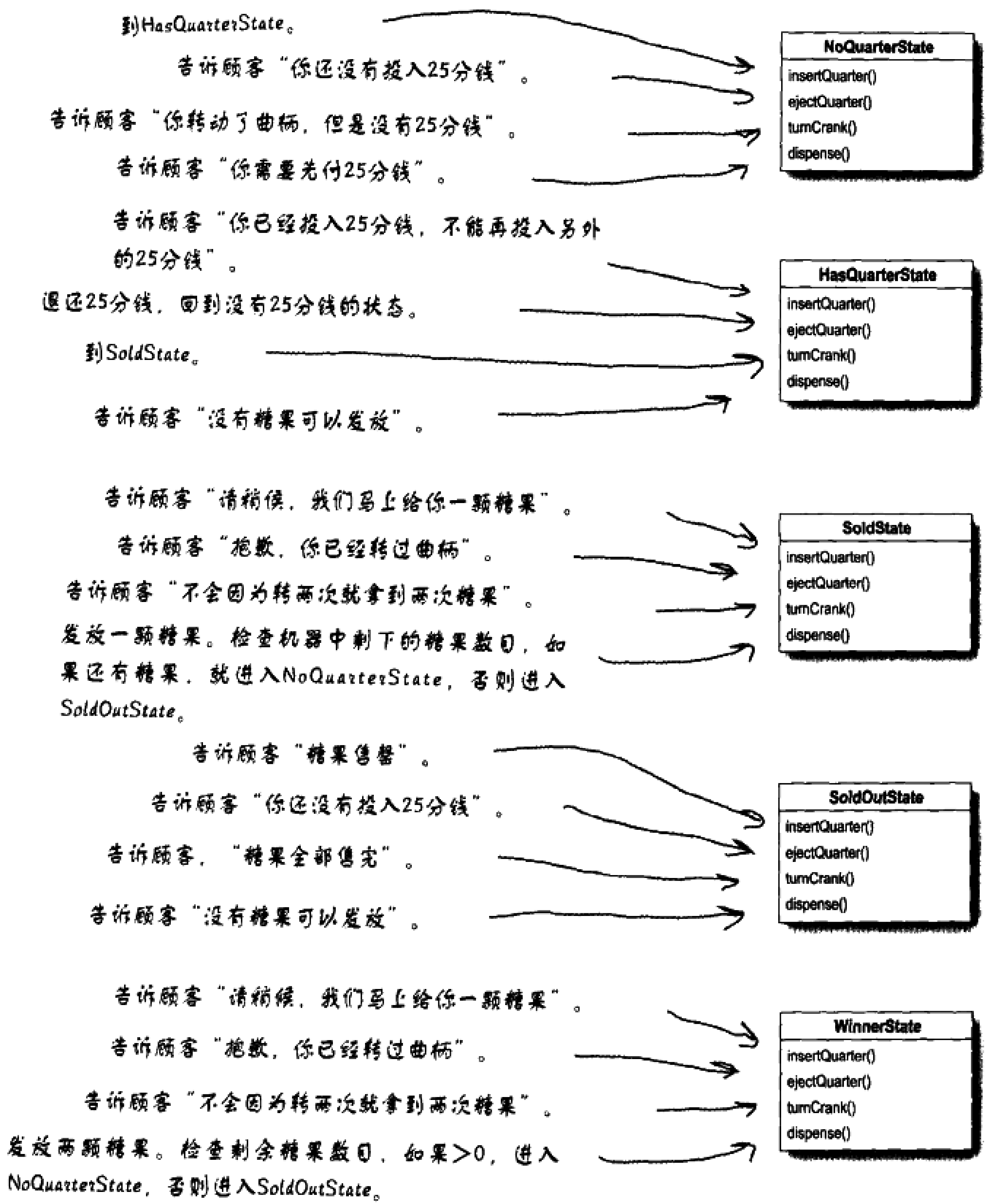
    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

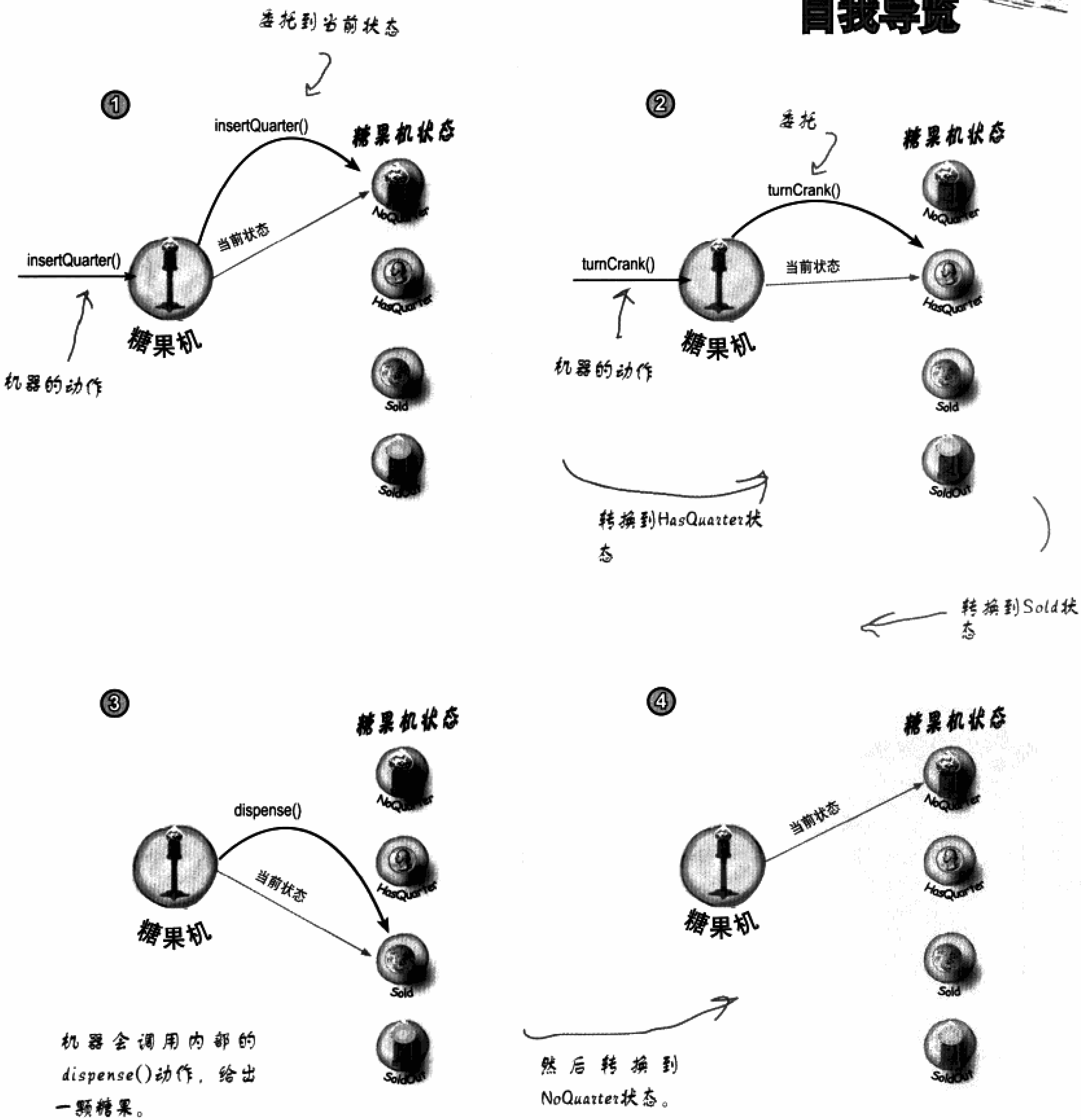
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

Sharpen your pencil

想要实现状态，我们首先需要指定当每一个动作被调用时，类的行为是哪一个。请在下面这张图上，为每个类的每个动作的行为加上注释。我们已经先帮你填写了其中的几个。



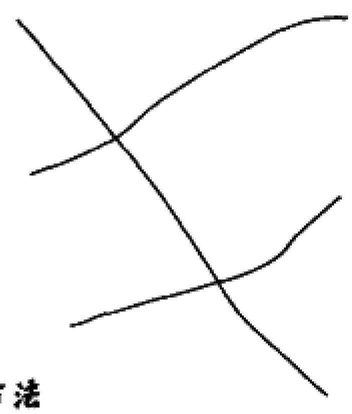
幕后花絮：自我导览


委托到当前状态
1
insertQuarter()
糖果机状态
insertQuarter()
当前状态
NoQuarter
HasQuarter
Sold
SoldOut
糖果机
机器的动作
2
委托
turnCrank()
糖果机状态
turnCrank()
当前状态
糖果机
机器的动作
转换到HasQuarter状态
转换到Sold状态
3
糖果机状态
dispense()
当前状态
糖果机
机器会调用内部的dispense()动作，给出一颗糖果。
4
糖果机状态
当前状态
糖果机
然后转换到NoQuarter状态。

连连看

请将下列模式和描述配对

模式	描述
状态	将可以互换的行为封装起来，然后使用委托的方法，决定使用哪一个行为
策略	由子类决定如何实现算法中的某些步骤
模板方法	封装基于状态的行为，并将行为委托到当前状态



Sharpen your pencil

我们需要你为糖果机写一个refill()方法。这个方法需要一个变量——所要填入机器中的糖果数目。它应该能更新糖果机内的糖果数目，并重置机器的状态。

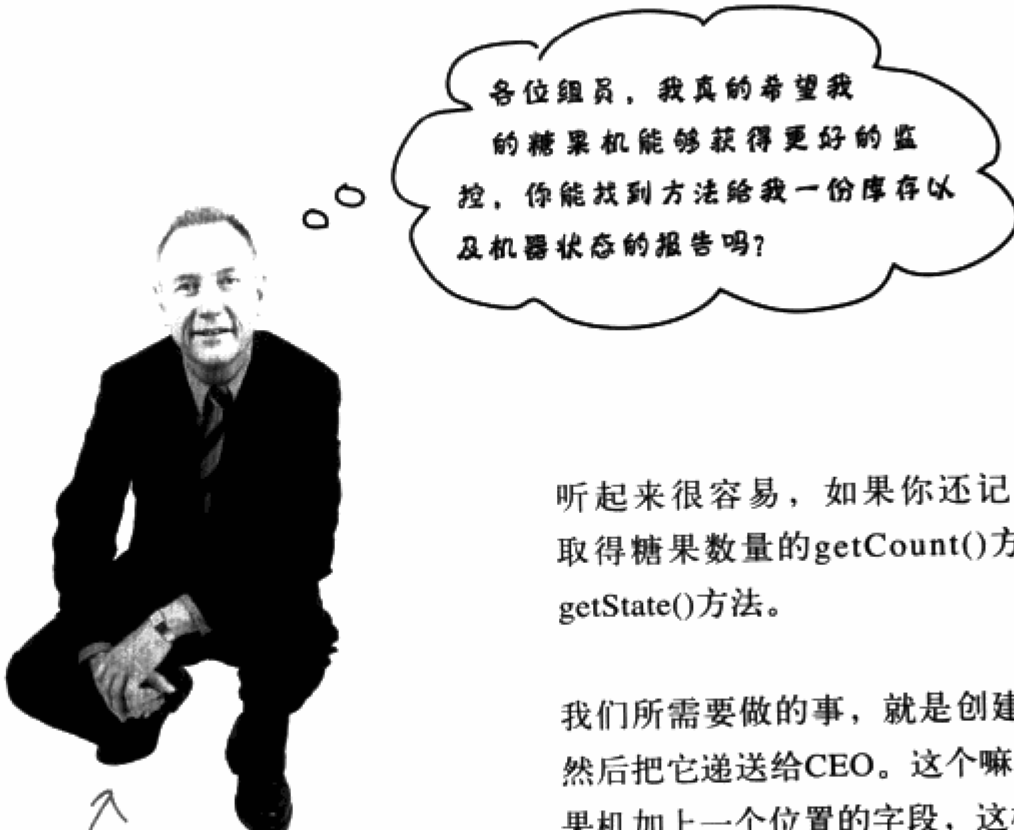
```
void refill(int count) {
    this.count = count;
    state = noQuarterState;
}
```

11 代理模式

控制对象访问



玩过扮白脸、扮黑脸的游戏吗？ 你是一个白脸，提供很好且很友善的服务，但是你不希望每个人都叫你做事，所以找了黑脸控制对你的访问。这就是代理要做的：控制和管理访问。就像你将看到的，代理的方式有许多种。代理以通过Internet为它们的代理对象搬运的整个方法调用而出名，它也可以代替某些懒惰的对象做一些事情。



还记得我吧！我就是万能糖果公司的CEO。

听起来很容易，如果你还记得我们已经得到了可以取得糖果数量的getCount()方法和取得糖果机状态的getState()方法。

我们所需要做的事，就是创建一份能打印出来的报告，然后把它递送给CEO。这个嘛！我们可能需要为每个糖果机加上一个位置的字段，这样CEO就可以一目了然。

让我们现在就开始编码。这一定会让CEO印象深刻，让他对我们彻底改观。

为监视器编码

我们先为GumballMachine加上处理位置的支持：

```
public class GumballMachine {
    // 其他实例变量
    String location;

    public GumballMachine(String location, int count) {
        // 构造器内的其他代码
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // 其他方法
}
```

位置用String记录。

位置被传入构造器内，然后存到此实例变量中。

让我们也加上一个getter方法，以便在需要位置时可以取得。

现在让我们创建另一个类，GumballMonitor（糖果监视器），以便取得机器的位置、糖果的库存量以及当前机器的状态，并打印成一份可爱的报告。

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

此监视器的构造器需要被传入糖果机，它会将糖果机记录在machine实例变量中。

负责打印报告的report方法，会将位置、库存、机器状态打印出来。

测试监视器

我们一下就搞定了，CEO将对我们的开发能力感到折服。

现在我们需要实例化一个GumballMonitor（糖果监视器），并传入一个糖果机：

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        int count = 0;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        count = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        GumballMonitor monitor = new GumballMonitor(gumballMachine);

        // 其他的测试代码

        monitor.report();
    }
}
```

利用命令行传入位置和一开始的糖果数目。

别忘了将位置和数目传入构造器……

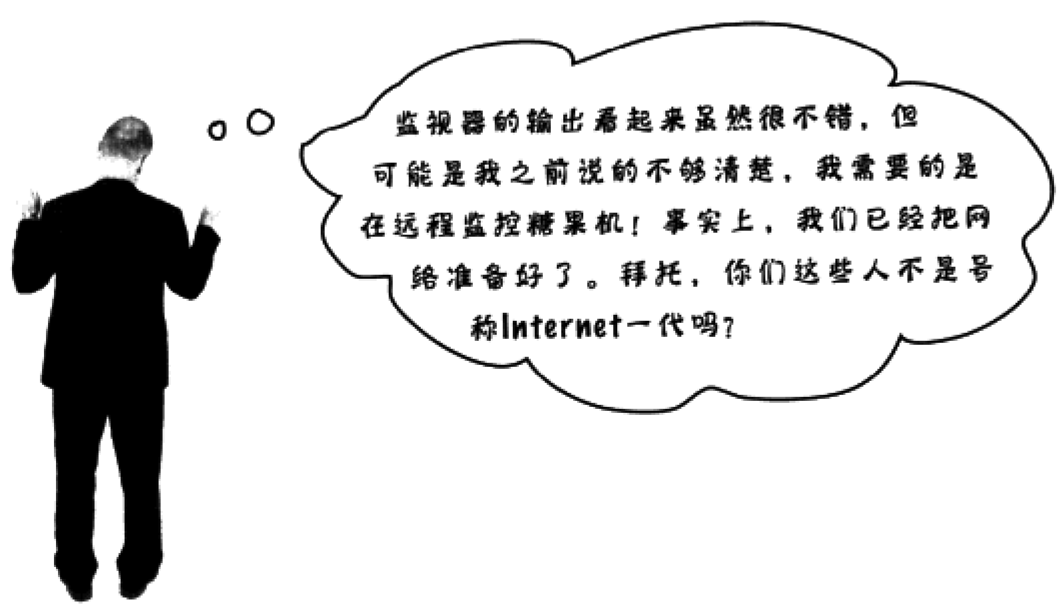
……然后实例化一个监视器，传给它一个机器来提供报告。

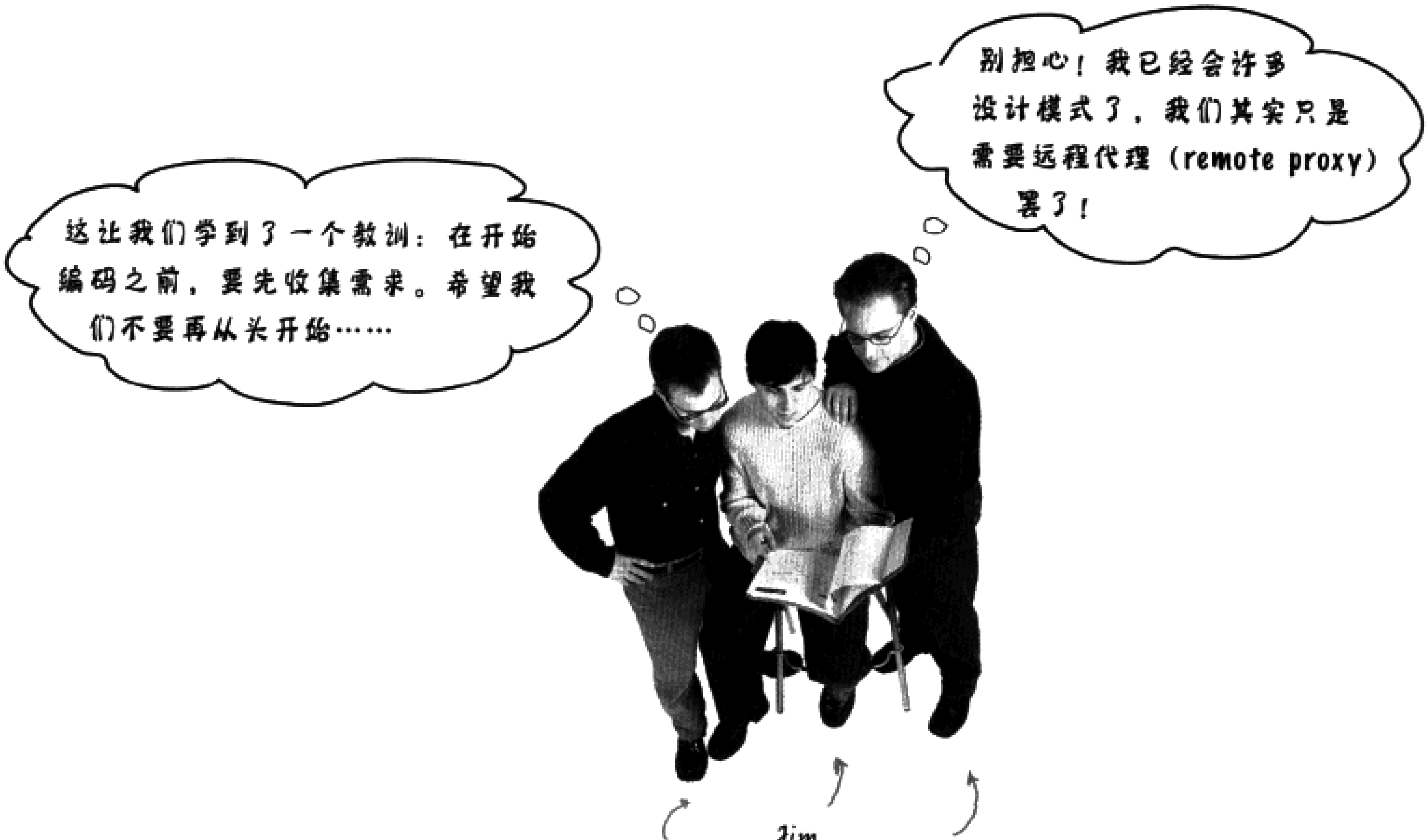
当我们需要机器的报告的时候，调用report()方法即可。

```
File Edit Window Help FlyingFish
%java GumballMachineTestDrive Seattle 112

Gumball Machine: Seattle
Current Inventory: 112 gumballs
Current State: waiting for quarter
```

输出是这样的！





Joe：你说远程什么？

Frank：远程代理。你想想：我们已经写好监视器代码，对吧？我们给GumballMonitor一个糖果机的引用，它给我们一份报告。问题在于监视器和糖果机在同一个JVM上面执行，但是CEO希望在他的桌面上远程监控这些机器！所以我们可以不要变化GumballMonitor，不要将糖果机交给GumballMonitor，而是将一个远程对象的代理交给它。

Joe：我不太懂。

Jim：我也不懂。

Frank：让我从头开始说……所谓的代理（proxy），就是代表某个真实的对象。在这个案例中，代理就像是糖果机对象一样，但其实幕后是它利用网络和一个远程的真正糖果机沟通。

Jim：你是说，不需要改我们的代码，只要将GumballMachine代理版本的引用交给监视器就可以了……

Joe：然后这个代理假装它是真正的对象，但是其实一切的动作是它利用网络和真正的对象沟通。

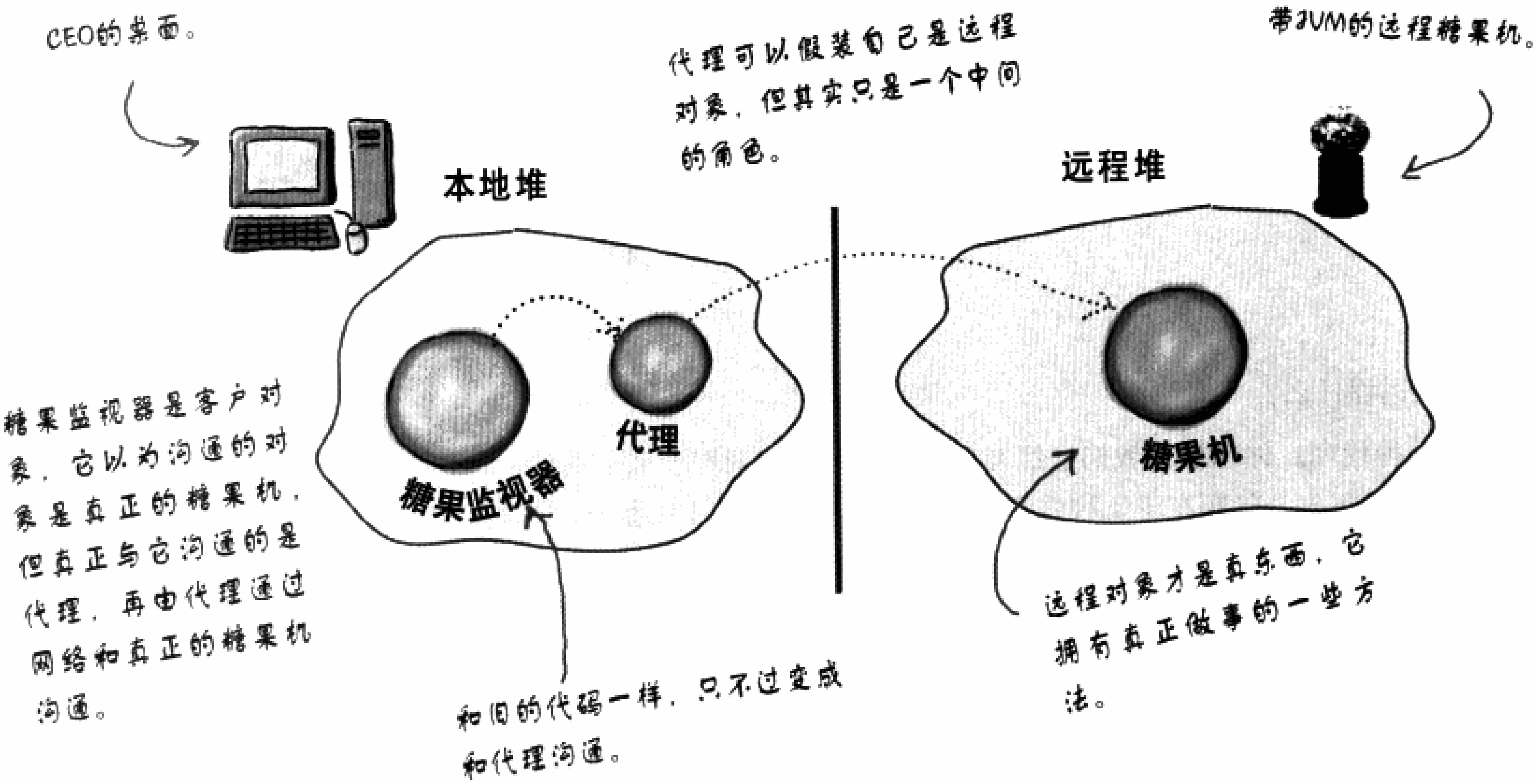
Frank：差不多就是这样。

Joe：这好像说的比做的容易。

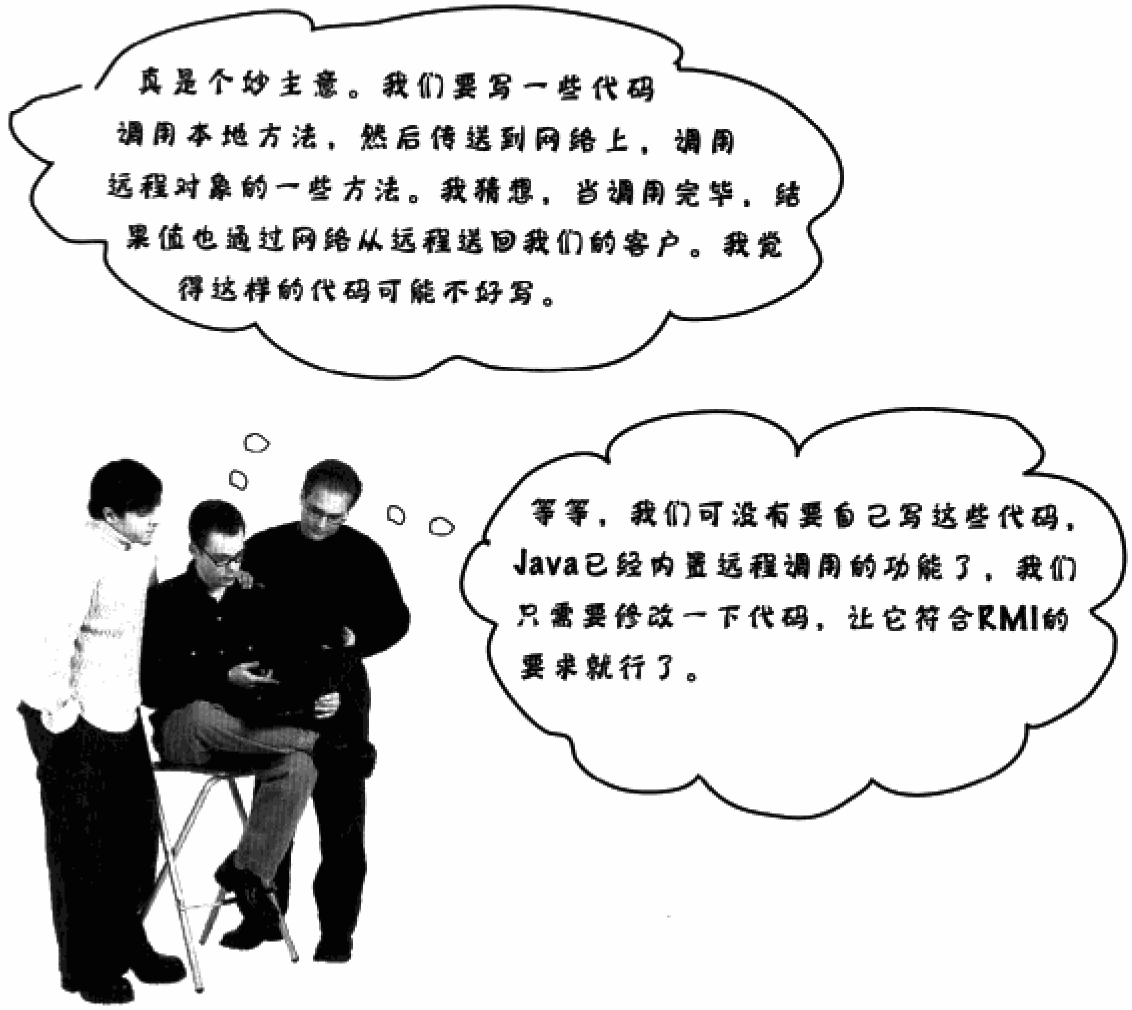
Frank：或许吧！但是我不认为有这么难。我们必须确定糖果机能够通过网络接受请求并且提供服务；我们也需要让监视器有办法取得代理对象的引用，这方面，幸好Java已经有一些很棒的内置工具可以帮助我们。我们先看看远程代理……

远程代理的角色

远程代理就好比“远程对象的本地代表”。何谓“远程对象”？这是一种对象，活在不同的Java虚拟机（JVM）堆中（更一般的说法为，在不同的地址空间运行的远程对象）。何谓“本地代表”？这是一种可以由本地方法调用的对象，其行为会转发到远程对象中。



你的客户对象所做的就像是在做远程方法调用，但其实只是调用本地堆中的“代理”对象上的方法，再由代理处理所有网络通信的低层细节。



BRAIN POWER

在我们进下一步之前，想想看要如何设计一个支持远程方法调用的系统。你要怎样才能让开发人员不用写太多代码？让远程调用看起来就好像本地调用一样，毫无瑕疵？

BRAIN² POWER

远程调用程序应该完全透明吗？这是个好主意吗？这个方法可能会产生问题吗？

将远程代理加到糖果机的监视代码中

构想上，这一切都很不错，但是要如何创建一个代理，知道如何调用在另一个JVM中的对象的方法？

这个嘛！你不能取得另一个堆的对象的引用，换句话说，你不可以这么写：

```
Duck d = <另一个堆的对象>
```

变量d只能引用当前代码语句的同一堆空间的对象。那该怎么办？该是Java远程方法调用出现的时刻了……RMI可以让我们找到远程JVM内的对象，并允许我们调用它们的方法。

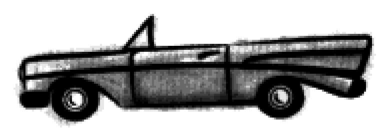
你可能在《Head First Java》书中看过RMI。如果你还不懂RMI，我们现在就稍微介绍一下，然后我们为糖果机代码添加代理支持。

我们打算这么做：

1. **首先，我们先浏览并了解一下RMI。即使你熟悉RMI，你可能还想复习顺便跟着浏览一下风景。**

2. **接着，我们会把GumballMachine变成远程服务，提供一些可以被远程调用的方法。**

3. **然后，我们将创建一个能和远程的GumballMachine沟通的代理，这需要用到RMI。最后再结合监视系统，CEO就可以监视任何数量的远程糖果机了。**



RMI浏览

如果你是RMI新手，仔细阅读下面几页。否则快速地扫一下就可以了。

远程方法101

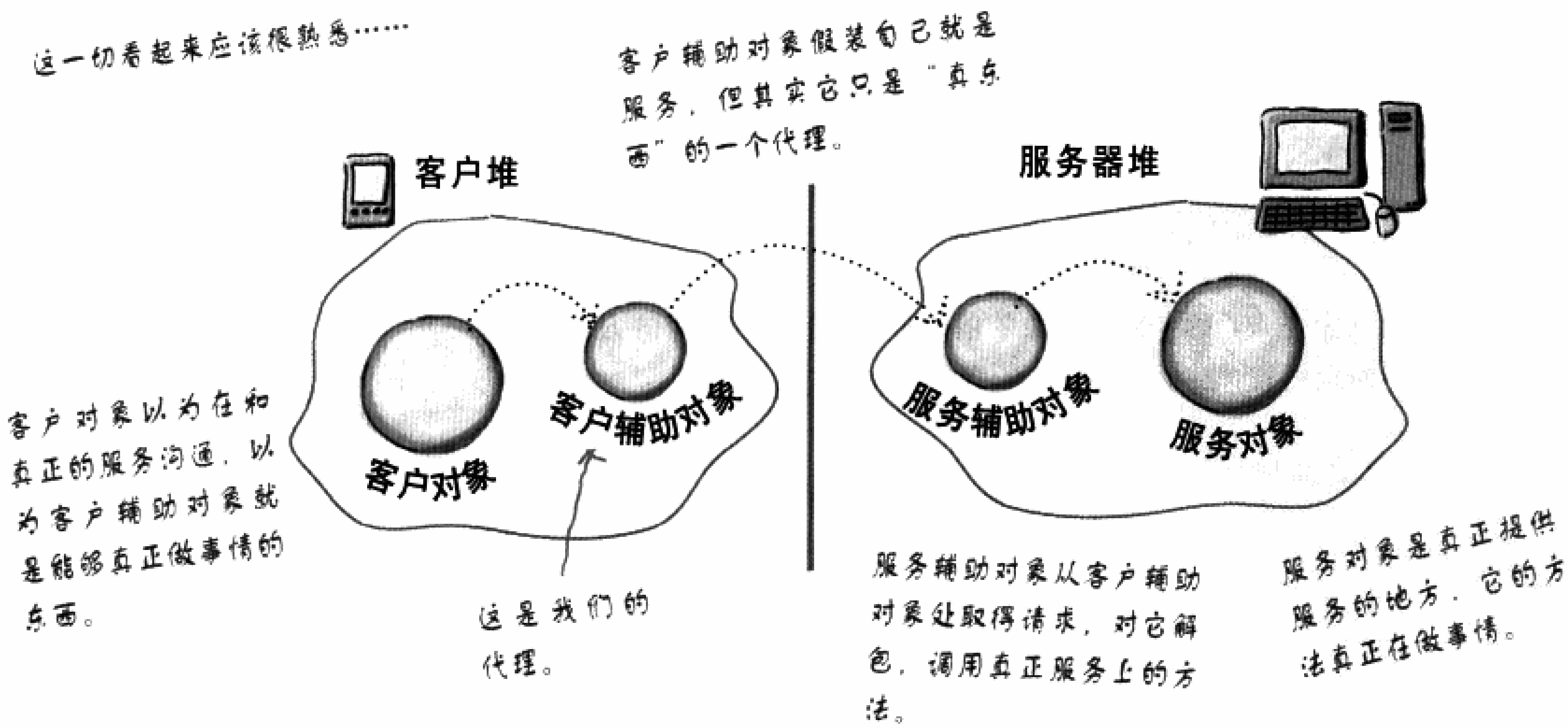
假如我们想要设计一个系统，能够调用本地对象，然后将每个请求转发到远程对象上进行。要如何设计？我们需要一些辅助对象，帮我们真正进行沟通。这些辅助对象使客户就像在调用本地对象的方法（事实也是如此）一样。客户调用客户辅助对象上的方法，仿佛客户辅助对象就是真正的服务。客户辅助对象再负责为我们转发这些请求。

换句话说，客户对象以为它调用的是远程服务上的方法，因为客户辅助对象乔装成服务对象，假装自己有客户所要调用的方法。

但是客户辅助对象不是真正的远程服务。虽然操作看起来很像（因为具有服务所宣称的相同的方法），但是并不真正拥有客户所期望的方法逻辑。客户辅助对象会联系服务器，传送方法调用信息（例如，方法名称、变量等），然后等待服务器的返回。

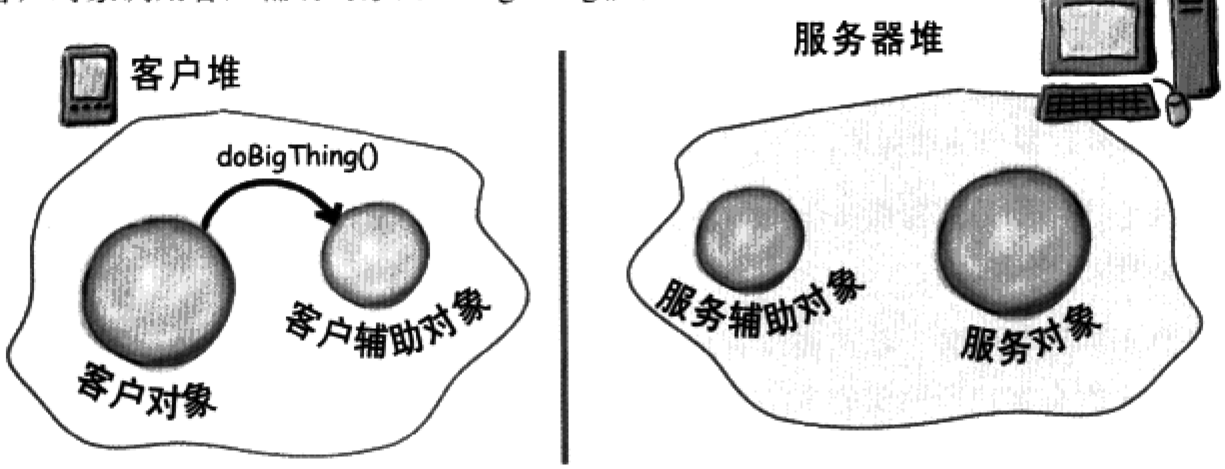
在服务器端，服务辅助对象从客户辅助对象中接收请求（透过Socket连接），将调用的信息解包，然后调用真正服务对象上的真正方法。所以，对于服务对象来说，调用是本地的，来自服务辅助对象，而不是远程客户。

服务辅助对象从服务中得到返回值，将它打包，然后运回到客户辅助对象（通过网络Socket的输出流），客户辅助对象对信息解包，最后将返回值交给客户对象。

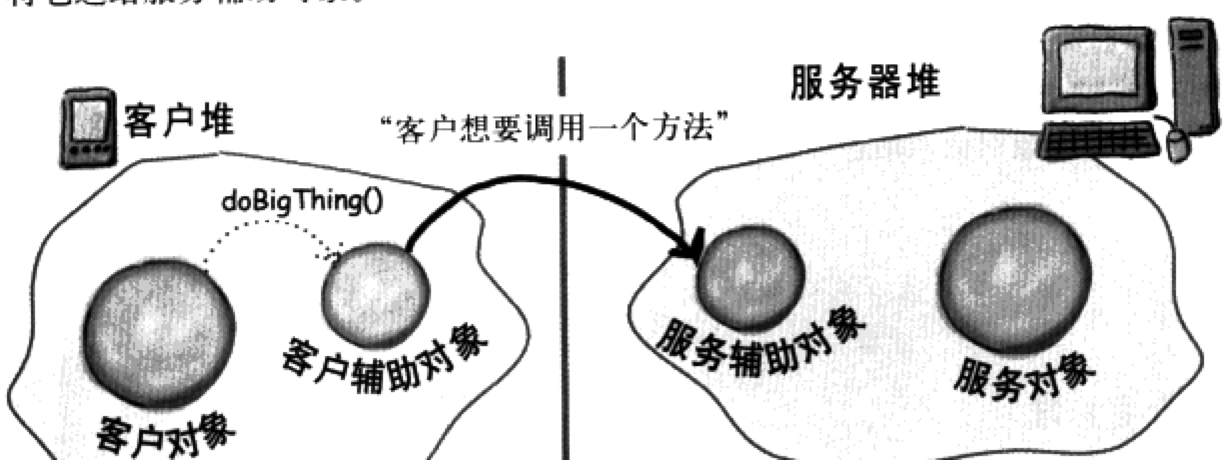


方法调用是如何发生的

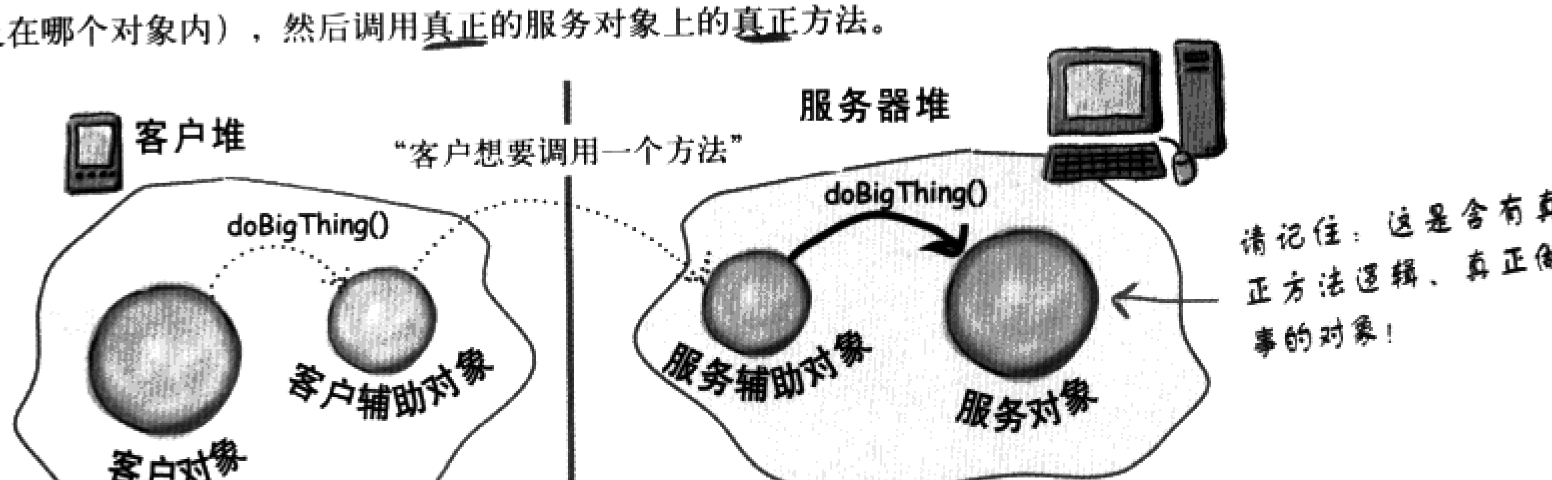
① 客户对象调用客户辅助对象的doBigThing()方法。



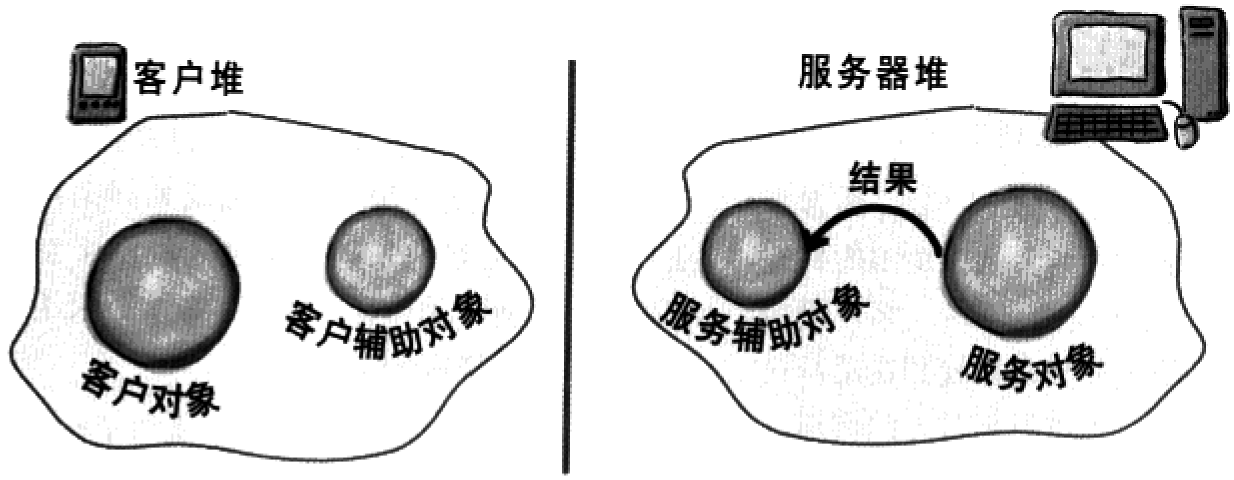
② 客户辅助对象打包调用信息（变量、方法名称等），然后通过网络将它运给服务辅助对象。



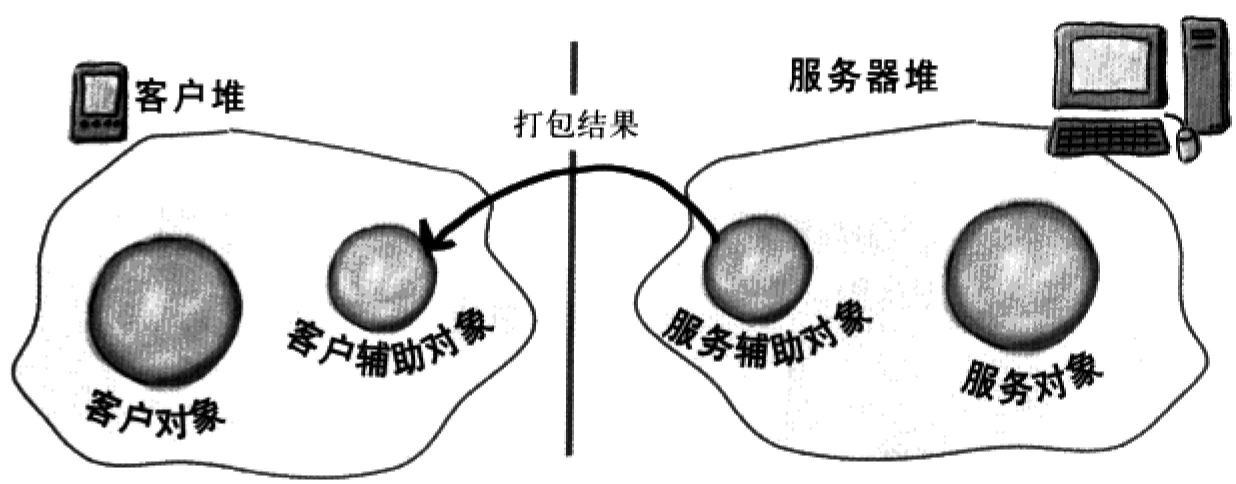
③ 服务辅助对象把来自客户辅助对象的信息解包，找出被调用的方法（以及在哪个对象内），然后调用真正的服务对象上的真正方法。



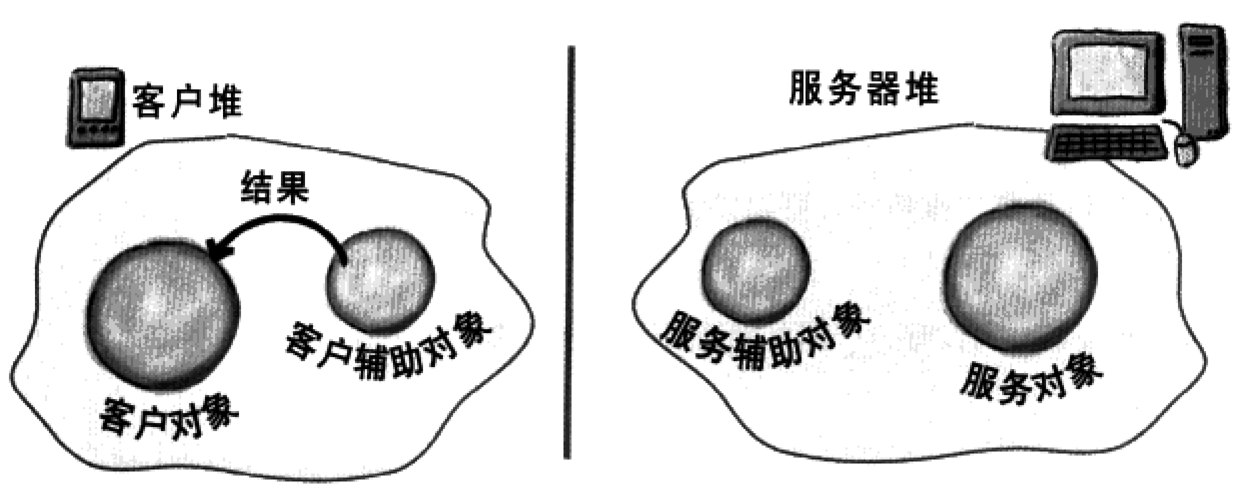
④ 服务对象上的方法被调用，将结果返回给服务辅助对象。



⑤ 服务辅助对象把调用的返回信息打包，然后通过网络运回给客户辅助对象。



⑥ 客户辅助对象把返回值解包，返回给客户对象。对于客户来说，这是完全透明的。



Java RMI概观

现在你已经知道远程方法如何工作的要点，你还需要了解如何利用RMI进行远程方法调用。

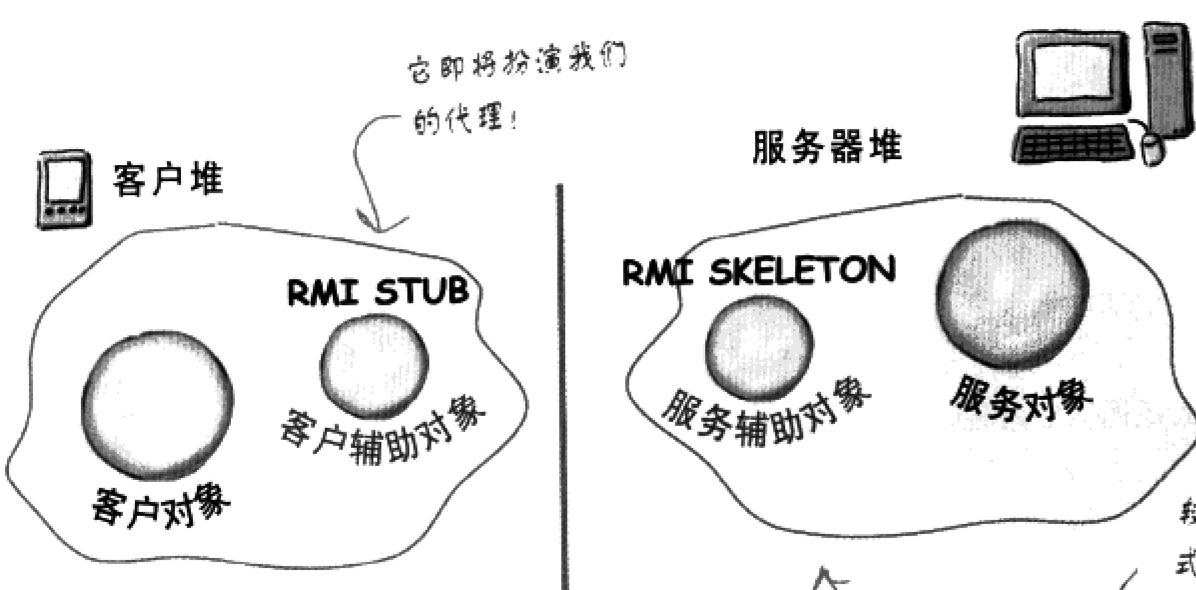
RMI提供了客户辅助对象和服务辅助对象，为客户辅助对象创建和服务对象相同的方法。RMI的好处在于你不必亲自写任何网络或I/O代码。客户程序调用远程方法（即真正的服务所在）就和在运行在客户自己的本地JVM上对对象进行正常方法调用一样。

RMI也提供了所有运行时的基础设施，好让这一切正常工作。这包括了查找服务（lookup service），这个服务用来寻找和访问远程对象。

关于RMI调用和本地（正常的）的方法调用，有一个不同点。虽然调用远程方法就如同调用本地方法一样，但是客户辅助对象会通过网络发送方法调用，所以网络和I/O的确是存在的。关于网络和I/O部分，我们知道些什么？

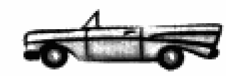
我们知道网络和I/O是有风险的，容易失败的，所以随时都可能抛出异常，也因此，客户必须意识到风险的存在。再过几页我们就会讨论这部分。

RMI称呼（译注：terminology，术语，重点在概念本身；nomenclature，称呼，重点在概念上贴的标签）：RMI将客户辅助对象称为stub（桩），服务辅助对象称为skeleton（骨架）。



现在，我们就来看看如何将对象变成服务——可以接受远程调用的服务。也看看，如何让客户做远程调用。

接下来会有一堆步骤和一些颠簸、大转弯……你可得系好安全带坐稳了，不过别太担心！



RMI浏览

制作远程服务

这里有用来制作远程服务的五个步骤的概要。换句话说，这些步骤将一个普通的对象变成可以被远程客户调用的远程对象。我们稍后会把这些步骤应用于GumballMachine。现在，就让我们看看这些步骤的细节。

步骤一：

制作远程接口

远程接口定义出可以让客户远程调用的方法。客户将用它作为服务的类类型。Stub和实际的服务都实现此接口。



MyService.java

接口定义了要供客户调用的远程方法。

步骤二：

制作远程的实现

这是做实际工作的类，为远程接口中定义的远程方法提供了真正的实现。这就是客户真正想要调用方法的对象（例如，我们的GumballMachine）。



MyServiceImpl.java

真正的服务。这个类具有方法，做真正的工作。它实现远程接口。

步骤三：

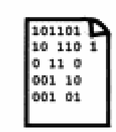
利用rmic产生的stub和skeleton。

这就是客户和服务的辅助类。你不需自己创建这些类，甚至连生成它们的代码都不用看，因为当你运行rmic工具时，这都会自动处理。你可以在JDK中找到rmic。

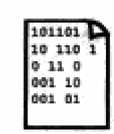
用rmic执行实现服务的类……

```
File Edit Window Help Eat
%rmic MyServiceImpl
```

……就会产生两个新的类，作为辅助对象。



MyServiceImpl_Stub.class



MyServiceImpl_Skel.class

步骤四：

启动RMI registry（rmiregistry）

rmireistry就像是电话簿，客户可以从中查到代理的位置（也就是客户的stub helper对象）。

```
File Edit Window Help Drink
%rmiregistry
```

在另一个终端上执行。

步骤五：

开始远程服务

你必须让服务对象开始运行。你的服务实现类会去实例化一个服务的实例，并将这个服务注册到RMI registry。注册之后，这个服务就可以供客户调用了。

```
File Edit Window Help BeMerry
%java MyServiceImpl
```

步骤一：制作远程接口

① 扩展java.rmi.Remote。

Remote是一个“记号”接口，所以Remote不具有方法。对于RMI来说，Remote接口具有特别的意义，所以我们必须遵守规则。请注意，我们这里说的是“扩展”（extends），因为接口可以“扩展”另一个接口。

```
public interface MyRemote extends Remote {
```

这表示此接口要用来支持远程调用。

② 声明所有的方法都会抛出RemoteException。

客户使用远程接口调用服务。换句话说，客户会调用实现远程接口的Stub上的方法，而Stub底层用到了网络和I/O，所以各种坏事情都可能会发生。客户必须认识到风险，通过处理或声明远程异常来解决。如果接口中的方法声明了异常，任何在接口类型的引用上调用方法的代码也必须处理或声明异常。

```
import java.rmi.*;

public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Remote接口在java.rmi中。

每次远程方法调用都必须考虑成是“有风险的”。在每个方法中声明RemoteException，可以让客户注意到这件事，并了解这可能是无法工作的。

③ 确定变量和返回值是属于原语（primitive）类型或者可序列化（Serializable）类型。

远程方法的变量和返回值，必须属于原语类型或Serializable类型。这不难理解。远程方法的变量必须被打包并通过网络运送，这要靠序列化来完成。如果你使用原语类型、字符串和许多API中内定的类型（包括数组和集合），都不会有问题。如果你传送自己定义的类，就必须保证你的类实现了Serializable。

如果你需要复习一下Serialbzable，可参考《Head First Java》。

```
public String sayHello() throws RemoteException;
```

这个返回值将从服务器经过网络运回给客户，所以必须是Serializable的。这样，才可将变量和返回值打包并传送。

步骤二：制作远程实现



RMI浏览

① **实现远程接口。**

你的服务必须实现远程接口，也就是客户将要调用的方法的接口。

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // 类中更多的代码
}
```

编译器会确认你已经实现了此接口所有的方法。在这个例子中，只有一个方法。

② **扩展UnicastRemoteObject。**

为了要成为远程服务对象，你的对象需要某些“远程的”功能。最简单的方式是扩展java.rmi.server.UnicastRemoteObject，让超类帮你做这些工作。

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

③ **设计一个不带变量的构造器，并声明RemoteException。**

你的新超类UnicastRemoteObject带来一个小问题：它的构造器抛出RemoteException。唯一解决这个问题的方法就是为你的远程实现声明一个构造器，这样就有了一个声明RemoteException的地方。当类被实例化的时候，超类的构造器总是会被调用。如果超类的构造器抛出异常，那么你只能声明子类的构造器也抛出异常。

```
public MyRemoteImpl() throws RemoteException { }
```

你不需要在构造器中放进任何代码，只需要有办法声明超类构造器会抛出异常。

④ **用RMI Registry注册此服务。**

现在你已经有一个远程服务了，必须让它可以被远程客户调用。你要做的是将此服务实例化，然后放进RMI registry中（记得先确定RMI Registry正在运行，否则注册会失败）。当注册这个实现对象时，RMI系统其实注册的是stub，因为这是客户真正需要的。注册服务使用了java.rmi.Naming类的静态rebind()方法。

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

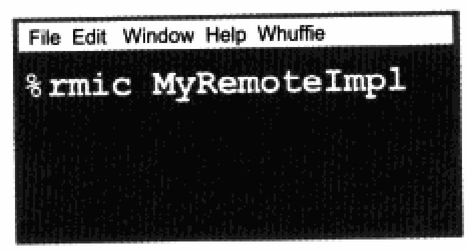
为你的服务命名，好让客户用来在注册表中寻找它，并在RMI registry中注册此名字和此服务。当你绑定（bind）服务对象时，RMI会把服务换成stub，然后把stub放到registry中。

步骤三：产生Stub和Skeleton

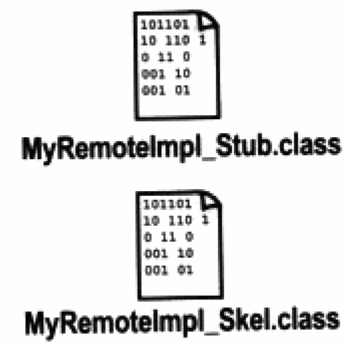
在远程实现类（不是远程接口）上执行rmic

rmic是JDK内的一个工具，用来为一个服务类产生stub和skeleton。命名习惯是在远程实现的名字后面加上_Stub或_Skel。rmic有一些选项可以调整，包括不要产生skeleton、查看源代码，甚至使用IIOP作为协议。我们这里使用rmic的方式是常用的方式，将类产生在当前目录下（就是你cd到的地方）。请注意，rmic必须看到你的实现类，所以你可能会从你的远程实现所在的目录执行rmic（为了简单起见，我们这里不用package。但是在真实世界中，你必须注意package的目录结构和名称问题）。

请注意，不需要在末尾加".class"，只要类名称就可以了。



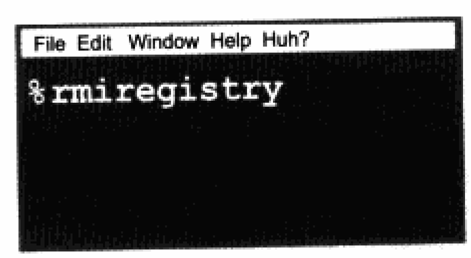
RMIC产生两个新类，作为辅助对象。



步骤四：执行remiregistry

开启一个终端，启动rmiregistry

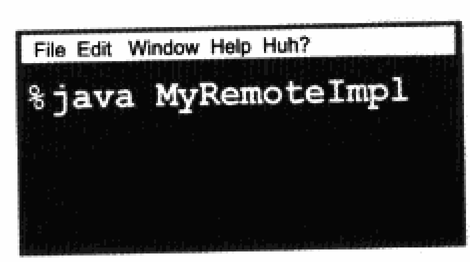
先确定启动目录必须可以访问你的类。最简单的做法是从你的"classes"目录启动。



步骤五：启动服务

开启另一个终端，启动服务

从哪里启动？可能是从你的远程实现类中的main()方法，也可能是从一个独立的启动类。在这个简单的例子中，我们是从实现类中的main()方法启动的，先实例化一个服务对象，然后到RMI registry中注册。



服务器端的完整代码

远程接口：

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

RemoteException和远程接口在java.rmi包中。

你的接口必须扩展java.rmi.Remote。

所有的远程方法都必须声明RemoteException。

远程服务（实现）：

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

   public String sayHello() {
       return "Server says, 'Hey'";
   }

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {

       try {
           MyRemote service = new MyRemoteImpl();
           Naming.rebind("RemoteHello", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

UnicastRemoteObject在java.rmi.server包中。

扩展UnicastRemoteObject是最容易的方法。想要创建一个远程对象，

你必须实现你的远程接口！

你必须实现远程接口。当然你必须要实现所有的接口方法，但请注意，不需要声明RemoteException。

你的超类（UnicastRemoteObject）构造器声明了异常，所以你必须写一个构造器，因为这意味着你的构造器正在调用不安全的代码（它的超构造器）。

先产生远程对象，再使用Naming.rebind()绑定到rmiregistry。客户将使用你所注册的名称在RMI registry中寻找它。

客户如何取得stub对象？

客户必须取得stub对象（我们的代理）以调用其中的方法。所以我们就需要RMI Registry的帮忙。客户从Registry中寻找（lookup）代理，就好像在电话簿里寻找一样，说："我要找这个名字的stub。"

我们现在就来看看那些我们需要寻找并取得某个stub对象的代码。

想知道这一切是怎么工作的，看这里。

再靠近一点

客户总是使用远程接口作为服务类型，事实上客户不需要知道远程服务的真正类名是什么。

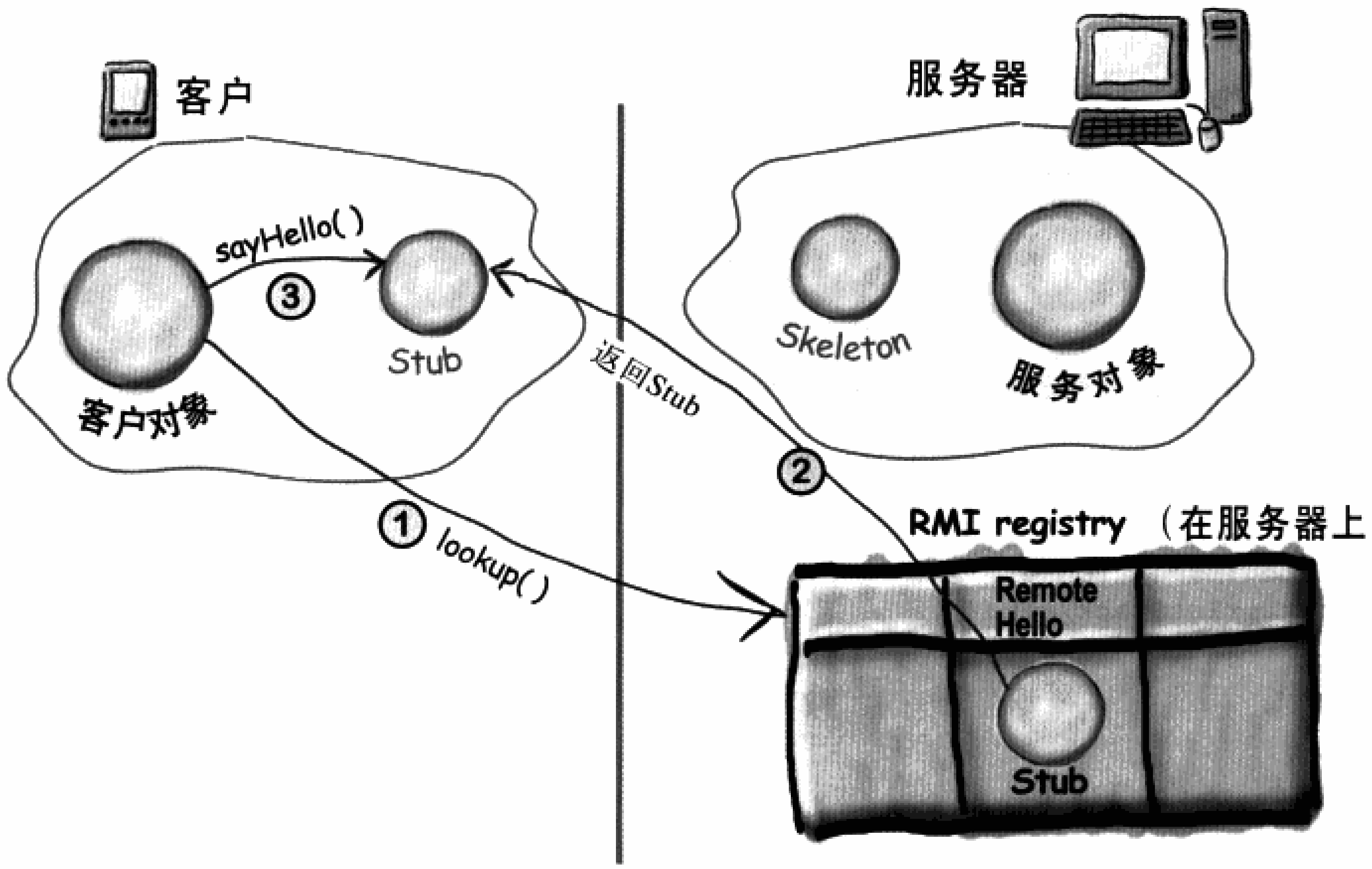
lookup()是Naming类的静态方法。

这必须是注册时用的名字。

```
MyRemote service =
    (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

lookup()的返回值是Object类型，你必须把它转成远程接口。

用来指出服务运行位置的主机名或IP地址。



工作方式……

① 客户到RMI registry中寻找。

```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

② RMI registry返回Stub对象。

（作为lookup方法的返回值）然后RMI会自动对stub反序列化。你在客户端必须有stub类（由rmic为你产生），否则stub就无法被反序列化。

③ 客户调用stub的方法，就像stub就是真正的服务对象一样。

完整的客户代码

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

  public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup( rmi://127.0.0.1/RemoteHello );

        String s = service.sayHello();

        System.out.println(s);
     } catch(Exception ex) {
         ex.printStackTrace();
     }
   }
}
```

用来做rmiregistry lookup 的Naming 类在java.rmi包中。

返回值是Object类型，所以别忘了转换类型。

你需要IP地址或主机名。

以及服务被绑定/重绑定时用的名称。

看起来和一般的老式方法调用没什么两样！（除了必须注意RemoteException之外。）

极客秘笈

客户如何取得stub类？

现在我们有一个有趣的问题。不管怎样，客户在做lookup时必须有stub类（之前利用rmic产生的），否则stub在客户端就无法被反序列化，一切也就告吹。客户端也需要调用远程对象方法所返回的序列化对象的类。如果是一个简单的系统，可以简单地把这些类移交到客户端。

还有一种更酷的方式，虽然超出本书范围，但是你可能会感兴趣，所以还是稍微提一下。这个酷方法是“动态类下载”（dynamic class downloading），利用动态类下载，序列化的对象（像stub）可以被“贴”上一个URL，告诉客户的RMI系统去寻找对象的类文件。在反序列化对象的过程中，如果RMI没有在本地发现类，就会利用HTTP的GET从该URL取得类文件。所以你需要一个简单的Web服务器来提供这些类文件，也需要更改客户端的安全参数。关于动态类下载，还有一些值得注意的主题，但是我们这里只是简述一下。

特别对于stub对象，客户还有另外一种方法可以取得类，但是只有Java 5才支持。我们会在本章末尾说明。





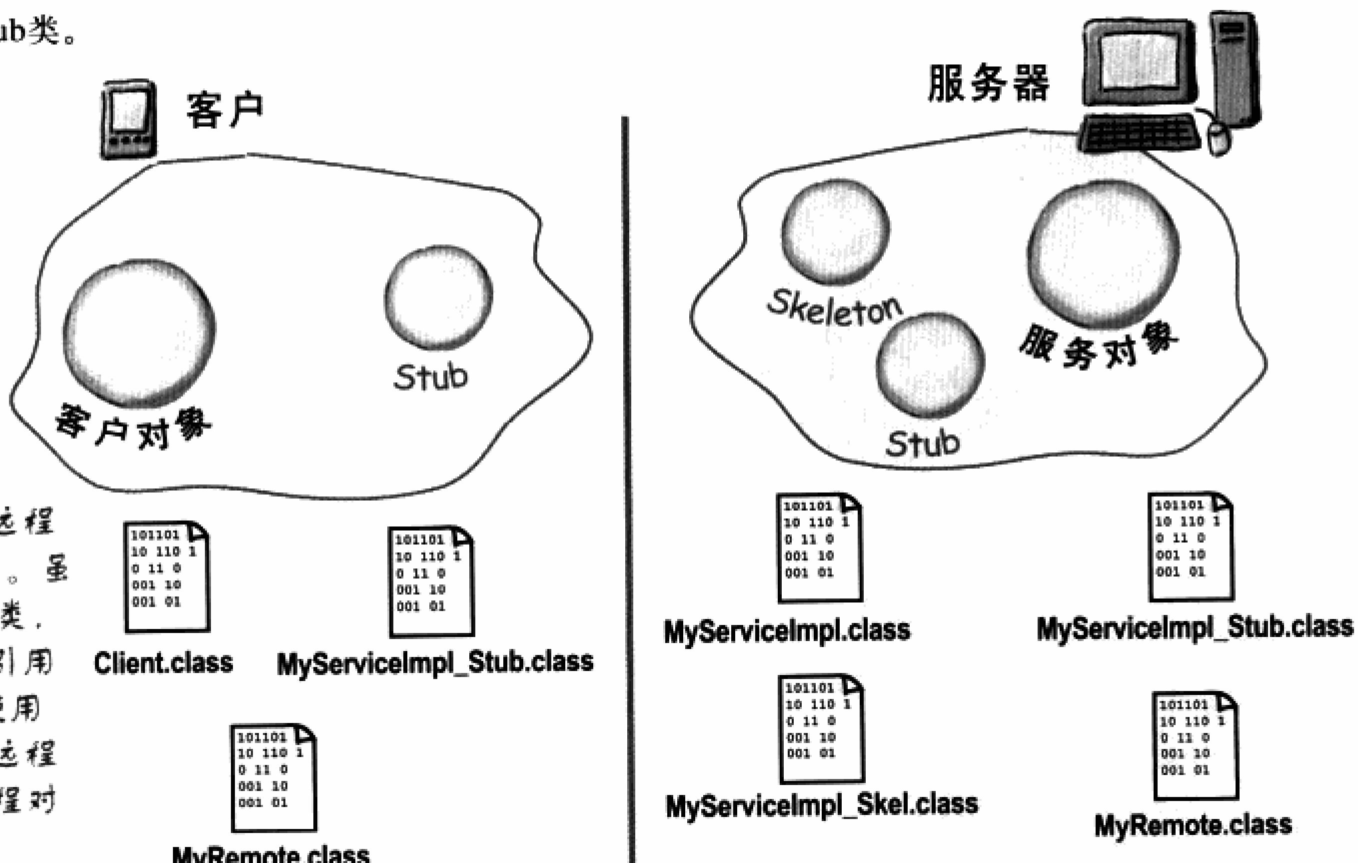
注意！

对于RMI，程序员最常犯的三个错误是：

1)忘了在启动远程服务之前先启动rmiregistry（要用Naming.rebind()注册服务，rmiregistry必须是运行的）。

2)忘了让变量和返回值的类型成为可序列化的类型（这种错误无法在编译期发现，只会在运行时发现）。

3)忘了给客户提供stub类。



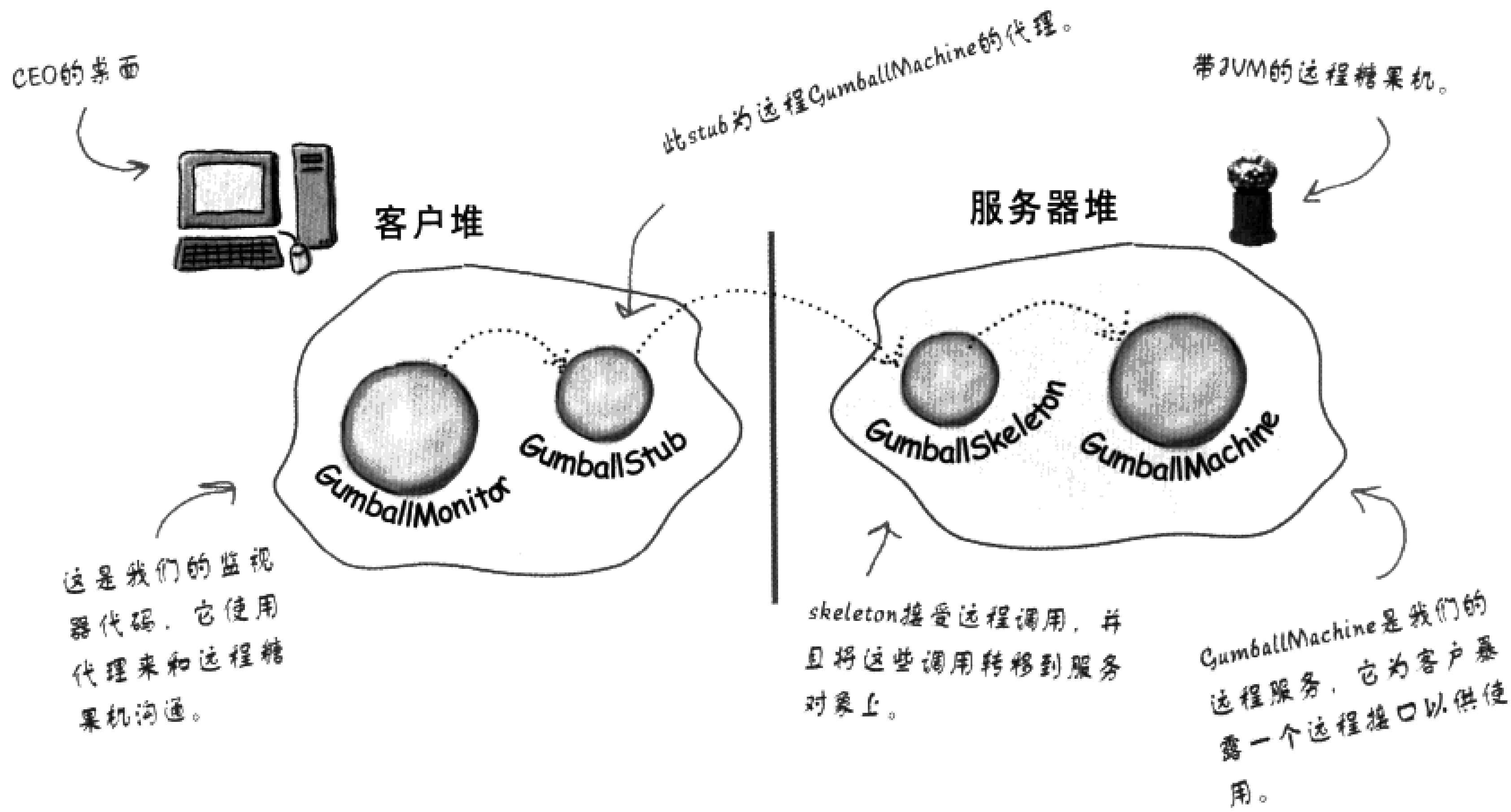
别忘了，客户使用远程接口调用stub的方法。虽然客户JVM需要stub类，但从来不在代码中引用stub类。客户总是使用远程接口，就如同远程接口就是真正的远程对象一样。

服务器需要Stub和Skeleton类，也需要服务和远程接口。之所以会需要stub类，是因为stub是真正服务的替身，当真正服务被绑定到RMI registry时，其实真正被绑定的是stub。

回头讨论我们的GumballMachine远程代理

结束浏览

OK，我们已经有了RMI的基础知识，现在可以用RMI实现糖果机的远程代理了。我们来看看GumballMachine是如何套用RMI框架的：



让GumballMachine准备好当一个远程服务

要把我们的代码改成使用代理，第一个步骤是让GumballMachine变成可以接受远程调用。换句话说，我们要把它变成一个服务。做法如下：

1)为GumballMachine创建一个远程接口。该口提供了一组可以远程调用的方法。

2)确定接口的所有返回类型都是可序列化的。

3)在一个具体类中，实现此接口。

我们从远程接口开始：

别忘了import java.rmi.*

这就是远程接口。

```
import java.rmi.*;

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```

所有的返回类型都必须是原语类型或可序列化类型……

这是准备支持的方法，每个都要抛出RemoteException。

我们有一个返回类型不是可序列化的：State类，现在来修改一下……

Serializable在java.io包内。

```
import java.io.*;

public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

然后我们只要扩展Serializable接口（此接口没有方法）。现在所有子类中的State就可以在网络上传送了。

实际上，我们还没处理完Serializable。对于State，我们有一个问题。你可能记得，每个状态对象都维护着一个对糖果机的引用，这样一来，状态对象就可以调用糖果机的方法，改变糖果机的状态。我们不希望整个糖果机都被序列化并随着State对象一起传送。修正这点很容易：

```
public class NoQuarterState implements State {
    transient GumballMachine gumballMachine;

    // 其他方法在这里。
}
```

对于State的每个实现，我们都在GumballMachine实例变量前面加上transient关键字，这样就告诉JVM不要序列化这个字段。

我们已经实现了GumballMachine类，但是需要确定它可以当成服务使用，并处理来自网络上的请求。为了做到这一点，我们必须确定GumballMachine实现GumballMachineRemote接口。

首先我们需要import rmi包。

GumballMachine要继承UnicastRemoteObject，以成为一个远程服务。

GumballMachine也需要实现这个远程接口……

```
import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    // 这里有实例变量

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // 这里有代码
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }

    // 这里有其他的方法
}
```

……构造器需要抛出RemoteException，因为超类是这么做的。

不要怀疑，这里完全不需要改！

在RMI registry中注册……

糖果机服务已经完成了。现在我们要将它装上去，好开始接受请求。首先我们要确保将它注册到RMI registry中，好让客户可以找到它。

我们要加上一点点代码进行测试：

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;
        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

            gumballMachine =
                new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

首先，我们需要在实例化糖果的代码周围加上try/catch块，因为我们的构造器可能抛出异常。

我们也添加上对Naming.rebind的调用，用gumballmachine的名字发布GumballMachine的stub。

让我们开始执行……

先执行这个。

这会启动并运行RMI registry服务。

```
File Edit Window Help Huh?
% rmiregistry
```

```
File Edit Window Help Huh?
% java GumballMachineTestDrive seattle.mightygumball.com 100
```

再执行这个。

这会使得GumballMachine启动和运行，并注册到RMI registry中

现在是GumballMonitor客户端……

还记得GumballMonitor吗？我们要在不改写它的情况下复用它，以符合网络的情况。为此，我们必须做一些小改变。

我们需要import RMI的包，因为下面将用到RemoteException类……

现在我们准备依赖此远程接口，而不是具体的GumballMachine类。

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

当我们试图调用那些最终要通过网络发生的方法时，我们需要捕获所有可能发生的远程异常。



编写监视器测试程序

现在我们已经具备所需要的一切，只需再写一些代码，让CEO可以监控许多糖果机。

这就是监视器测试程序，CEO会执行此程序！

被监视的位置有这些。

我们创建一个数组，数组内的元素是每台机器的位置。

```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://seattle.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0;i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                        (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for(int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

我们也创建监视器的数组。

现在，需要为每个远程机器创建一个代理。

然后我们遍历每台机器，将报告打印出来。

再靠近一点

这会返回一个远程糖果机的代理（如果无法定位会抛出异常）。

Naming.lookup()是RMI包内的静态方法，它从参数中得知位置和服务名称，然后在该位置的rmiregistry中寻找该名称的服务。

```
try {
    GumballMachineRemote machine =
            (GumballMachineRemote) Naming.lookup(location[i]);

    monitor[i] = new GumballMonitor(machine);

} catch (Exception e) {
    e.printStackTrace();
}
```

一旦有了远程机器的代理，我们就可以创建一个新的GumballMonitor，把要监视的机器传给它。

为万能糖果公司CEO准备的另一个展示……

现在，让我们把所有这些放在一起，进行另一个展示。首先，确定有一些新版的糖果机正在执行新代码：

每部机器上，在后台或者在另一个终端窗口执行rmiregistry……

……然后执行GumballMachine，指定它的位置和初始的糖果数目。

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive santafe.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive boulder.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive seattle.mightygumball.com 250
```

畅销的机器，糖果多放一些。

接着，我们将监视器交到CEO手上，希望这次他会喜欢：

```
File Edit Window Help GumballsAndBeyond
com
Current inventory: 99 gumballs
Current state: waiting for quarter

Gumball Machine: boulder.mightygumball.com
Current inventory: 44 gumballs
Current state: waiting for turn of cranl

Gumball Machine: seattle.mightygumball.c
Current inventory: 187 gumballs
Current state: waiting for quarter
%
```

此监视器遍历各台远程机器，并调用它们的getLocation()、getCount()、getState()方法。

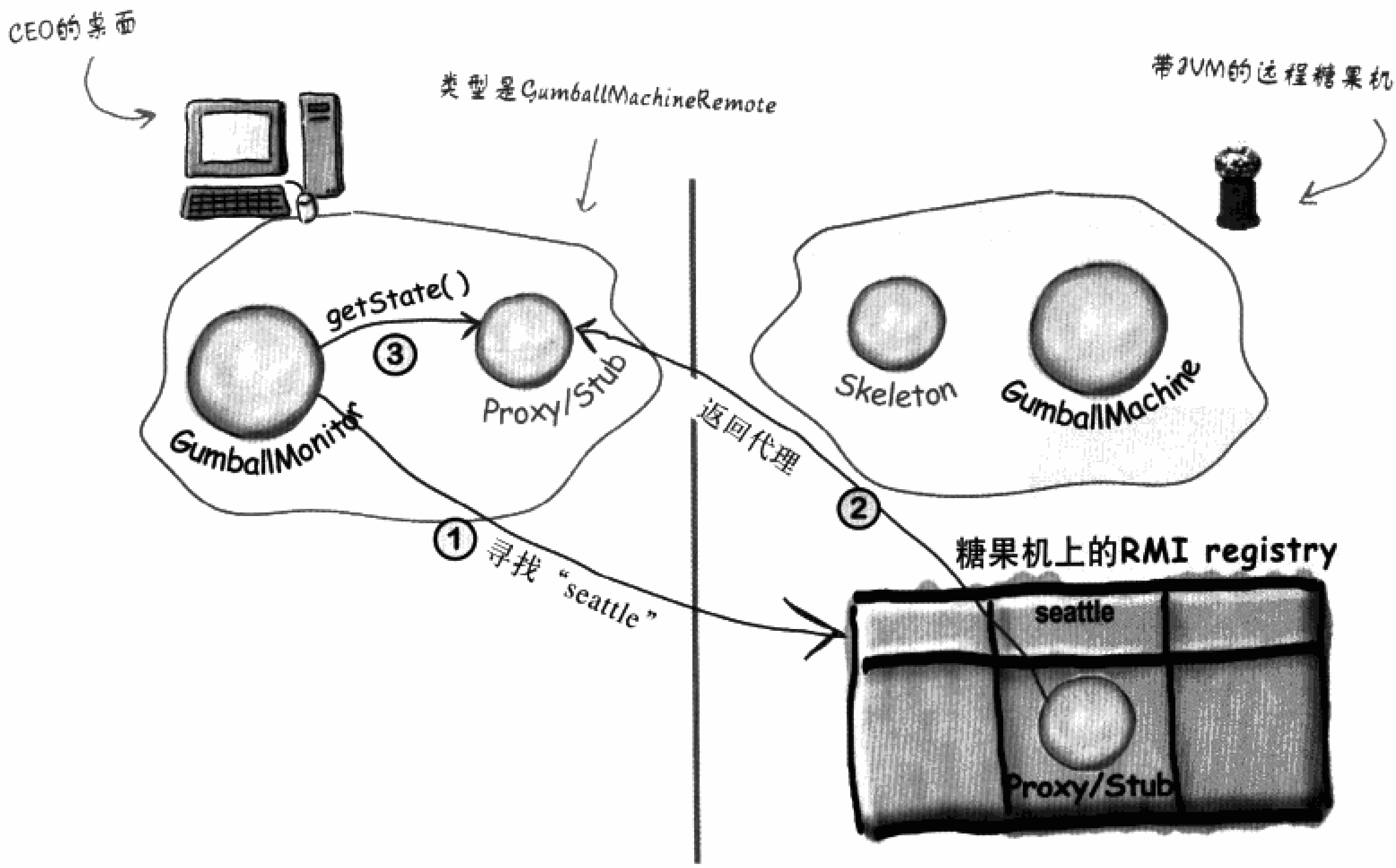


通过调用代理的方法，远程调用可以跨过网络，返回字符串、整数和State对象。因为我们使用的是代理，调用的方法会在远程执行，GumballMonitor根本就不知道/或不在乎这一点（唯一要操心的是：要处理远程异常）。

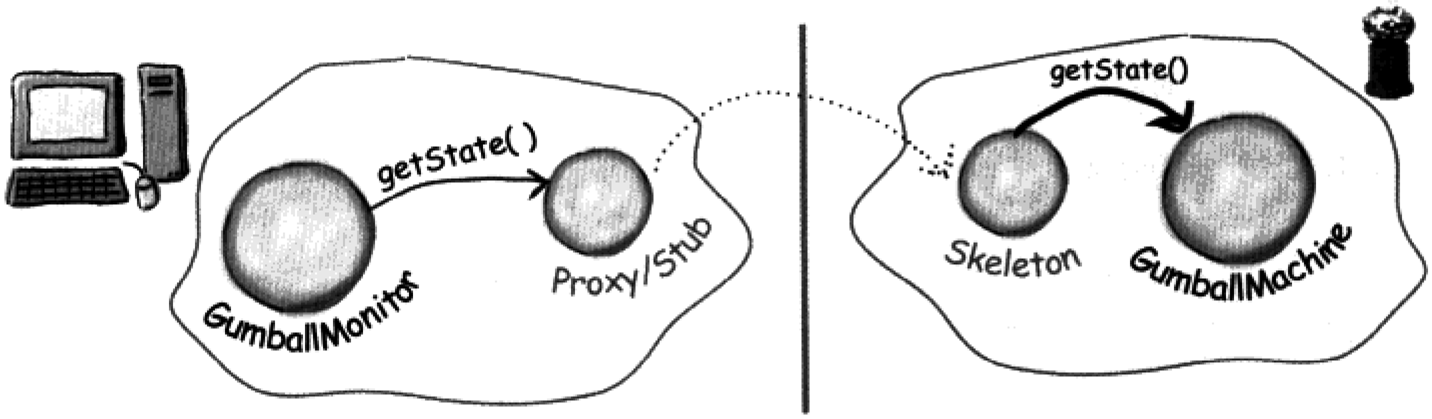




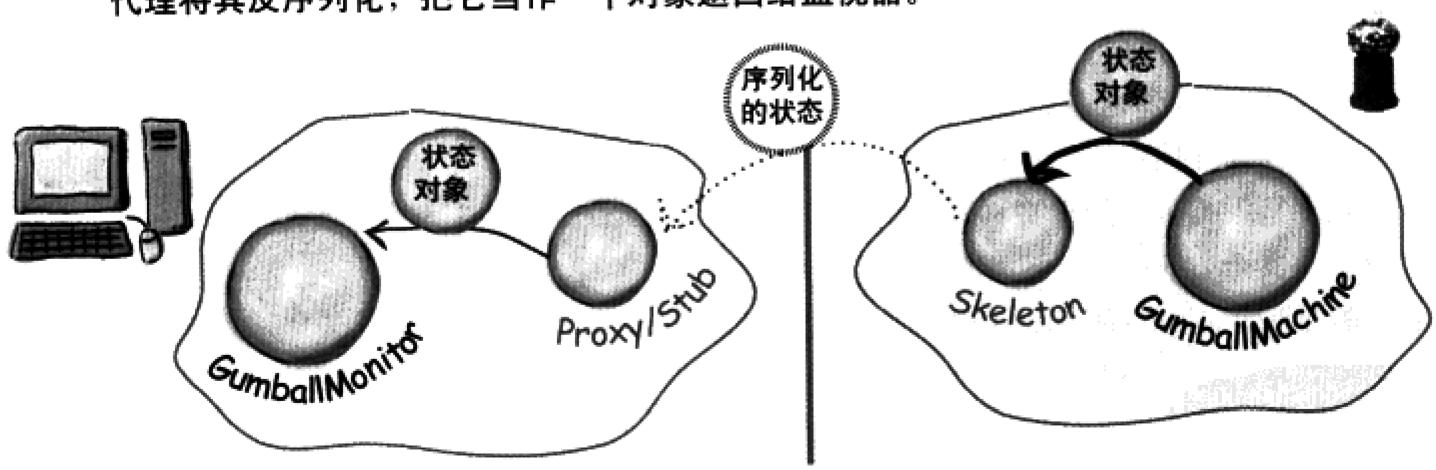
1. CEO执行监视器，先取得远程糖果机的代理，然后调用每个代理的getState()（以及getCount()和getLocation()）。



❷ 代理上的getState()被调用，此调用被转发到远程服务。Skeleton接收到请求，然后转发给糖果机。



❸ 糖果机将状态返回给skeleton，skeleton将状态序列化，通过网络传回给代理，代理将其反序列化，把它当作一个对象返回给监视器。



监视器除了知道它可能会遭遇远程异常之外根本没有改变。它也使用GumballMachineRemote接口，而不是具体的实现。

GumballMachine实现了另一个接口，而且可能在构造器中抛出远程异常。除此之外，糖果机的代码不需要改变。

我们也有一些代码负责使用RMI registry注册和定位stub，但是无论如何，如果我们需要在网络上工作，我们就需要这些定位服务。

定义代理模式

这一章的篇幅已经很大了，因为我们花了很多时间在解释远程代理。尽管如此，你还是会发现代理模式的定义和类图其实相当直接易懂。请注意，远程代理是一般代理模式的一种实现，其实这个模式的变体相当多，我们稍后会提到这些变体。

现在，我们就来看看代理模式的定义：

> **代理模式**为另一个对象提供一个替身或占位符以控制对这个对象的访问。

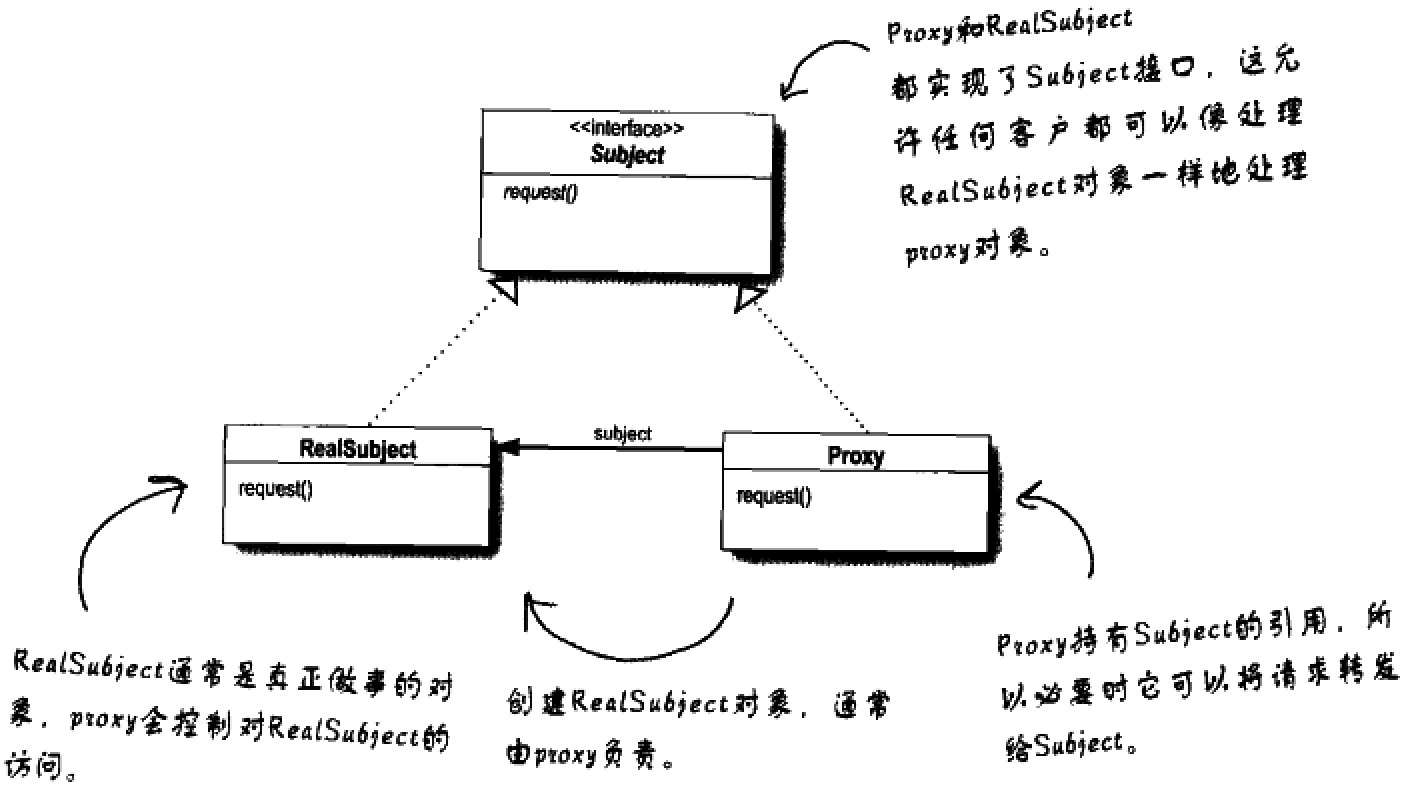
使用代理模式创建代表（representative）对象，让代表对象控制某对象的访问，被代理的对象可以是远程的对象、创建开销大的对象或需要安全控制的对象。

我们已经看到代理模式是如何为另一个对象提供替身的。我们也将代理描述成另一个对象的“代表”。

但是代理控制访问怎么解释呢？这听起来有一点奇怪。别担心，在糖果机的例子中，代理控制了对远程对象的访问。代理之所以需要控制访问，是因为我们的客户（监视器）不知道如何和远程对象沟通。从某个方面来看，远程代理控制访问，好帮我们处理网络上的细节。正如同刚刚说过的，代理模式有许多变体，而这些变体几乎都和“控制访问”的做法有关。稍后我们会对此讨论得更详细，目前我们还是先看看几种代理控制访问的方式：

- 就像我们已经知道的，远程代理控制访问远程对象。
- 虚拟代理控制访问创建开销大的资源。
- 保护代理基于权限控制对资源的访问。

现在你已经有基本的概念了，来看看类图……



让我们详细看这张图……

首先是Subject，它为RealSubject和Proxy提供了接口。通过实现同一接口，Proxy在RealSubject出现的地方取代它。

RealSubject是真正做事的对象，它是被proxy代理和控制访问的对象。

Proxy持有RealSubject的引用。在某些例子中，Proxy还会负责RealSubject对象的创建与销毁。客户和RealSubject的交互都必须通过Proxy。因为Proxy和RealSubject实现相同的接口（Subject），所以任何用到RealSubject的地方，都可以用Proxy取代。Proxy也控制了对RealSubject的访问，在某些情况下，我们可能需要这样的控制。这些情况包括RealSubject是远程的对象、RealSubject创建开销大，或RealSubject需要被保护。

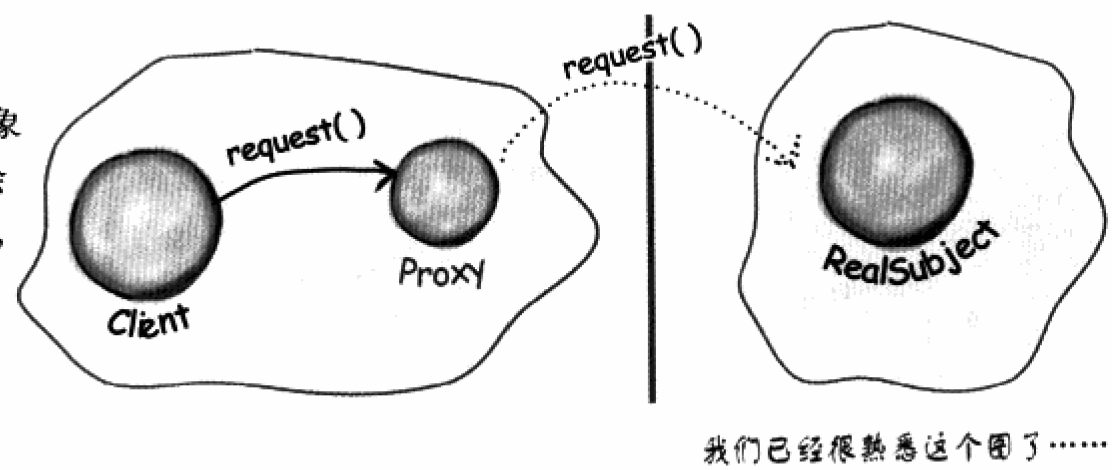
你已经了解了一般的代理模式，现在让我们看看，除了远程代理之外，代理模式还有哪些用法……

准备虚拟代理（Virtual Proxy）

你已经看过代理模式的定义，也看过一个特定的例子（远程代理），现在就让我们看看另一种代理：虚拟代理。你将发现，代理模式可以以很多形式显现，但都大致符合一般代理的设计。为何有这么多的形式呢？因为代理模式可以被用在许多不同的例子中。让我们现在看看虚拟代理和远程代理的比较：

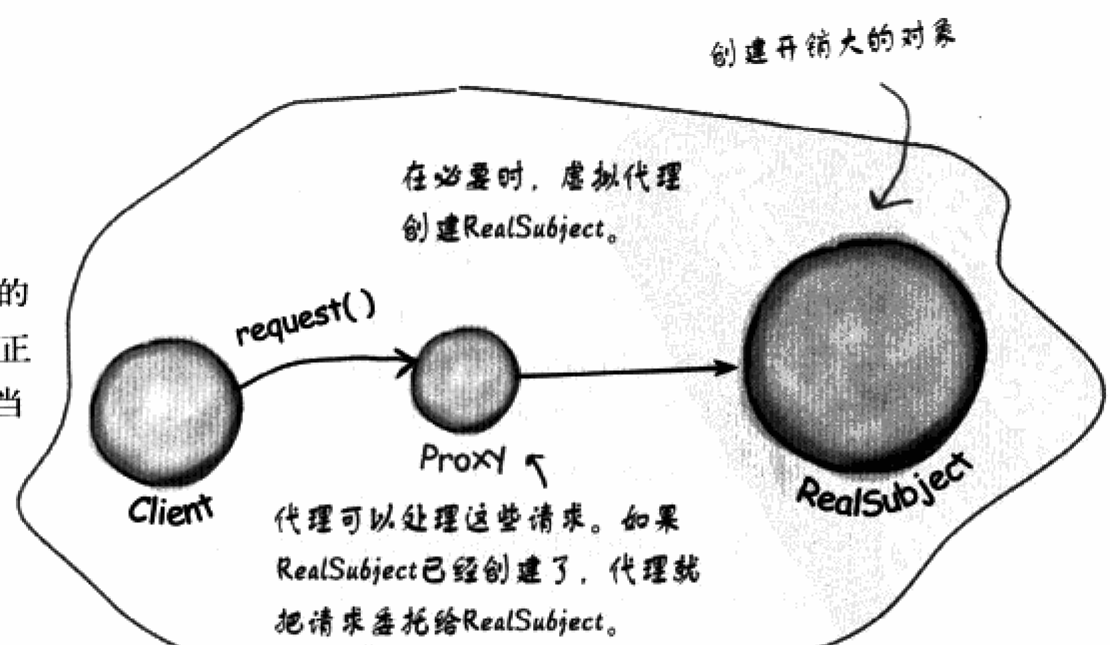
远程代理

远程代理可以作为另一个JVM上对象的本地代表。调用代理的方法，会被代理利用网络转发到远程执行，并且结果会通过网络返回给代理，再由代理将结果转给客户。



虚拟代理

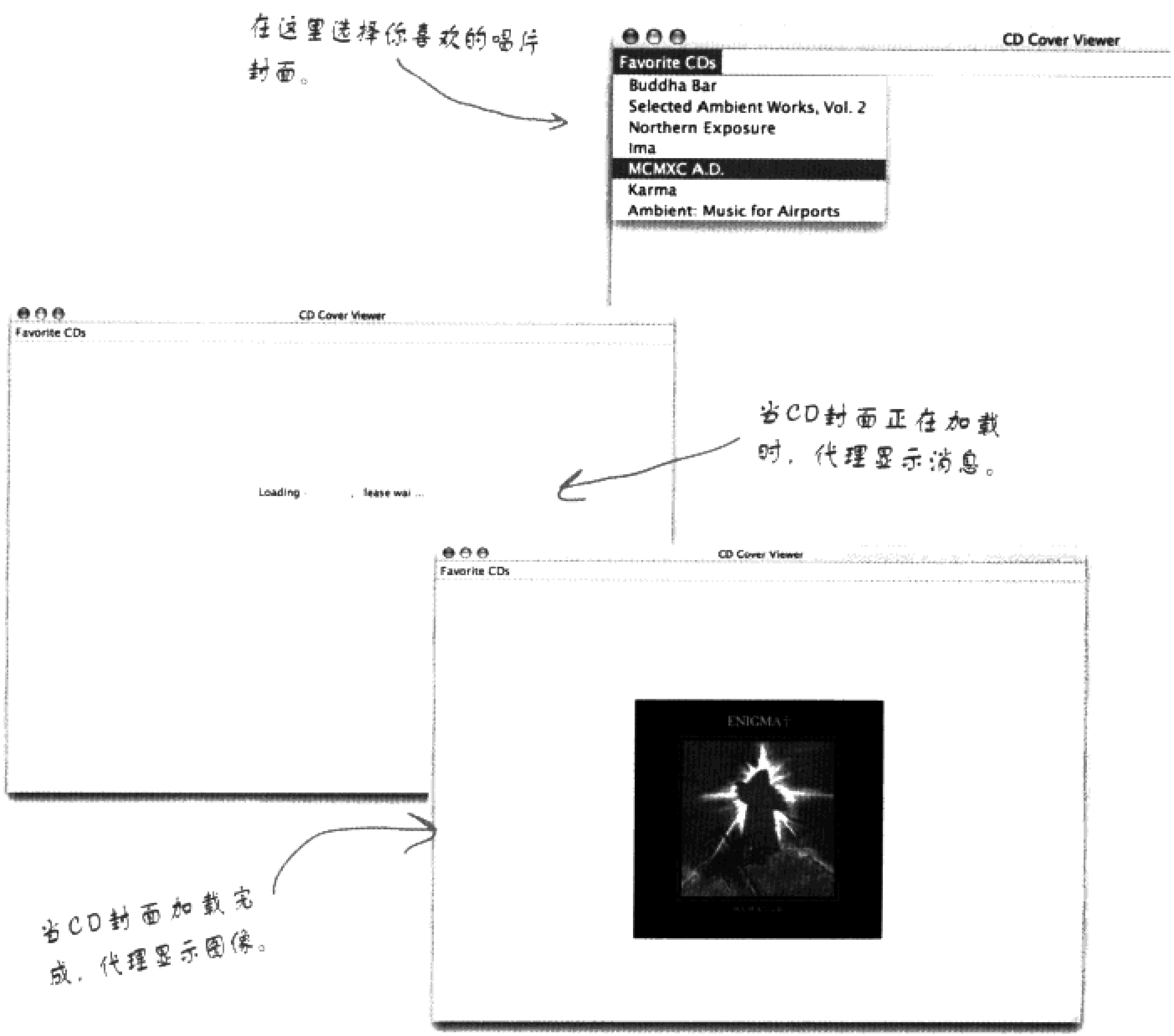
虚拟代理作为创建开销大的对象的代表。虚拟代理经常直到我们真正需要一个对象的时候才创建它。当对象在创建前和创建中时，由虚拟代理来扮演对象的替身。对象创建后，代理就会将请求直接委托给对象。



显示CD封面

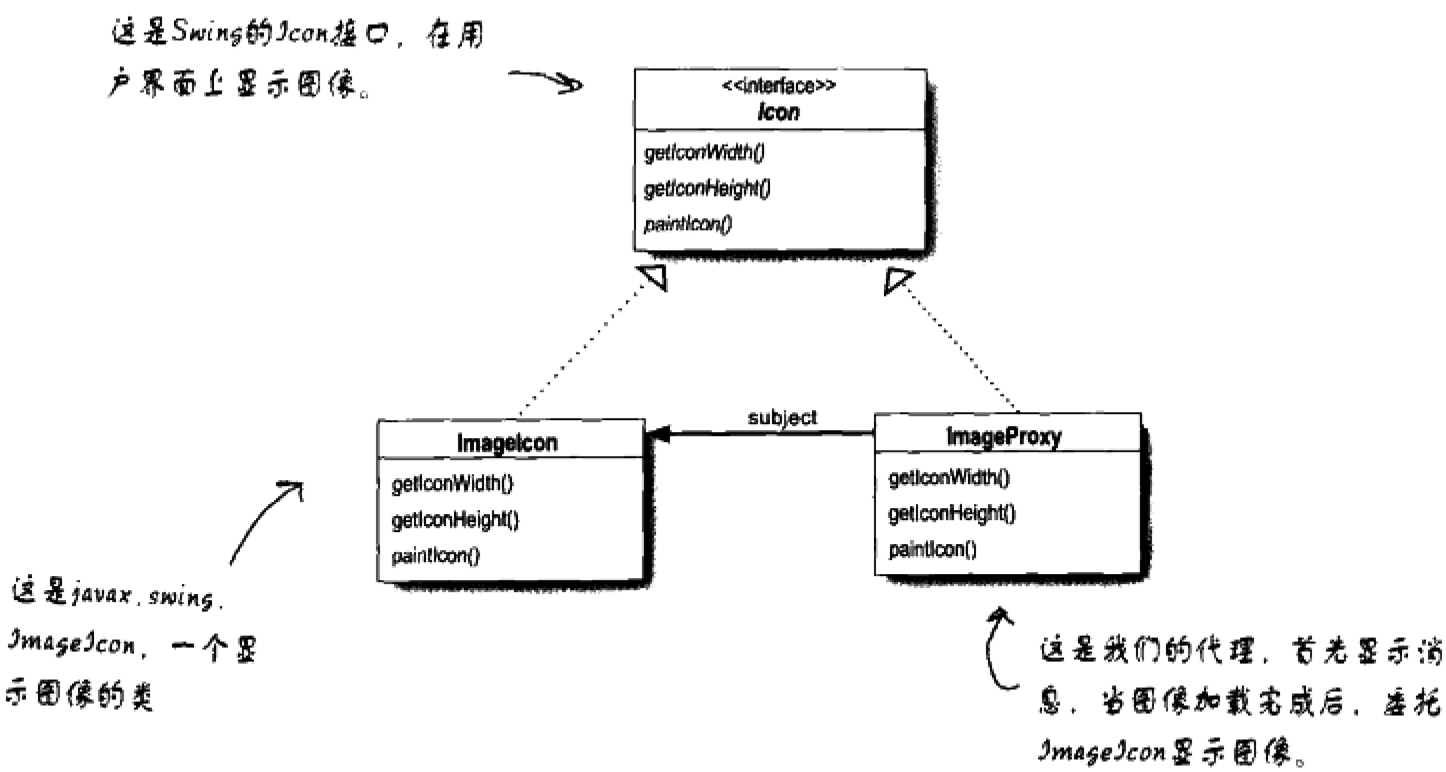
我们打算建立一个应用程序，用来展现你最喜欢的CD封面。你可以建立一个CD标题菜单，然后从Amazon.com等网站的在线服务中取得CD封面的图。如果你使用Swing，可以创建一个Icon接口从网络上加载图像。唯一的问题是，限于连接带宽和网络负载，下载可能需要一些时间，所以在等待图像加载的时候，应该显示一些东西。我们也不希望在等待图像时整个应用程序被挂起。一旦图像被加载完成，刚才显示的东西应该消失，图像显示出来。

想做到这样，简单的方式就是利用虚拟代理。虚拟代理可以代理Icon，管理背景的加载，并在加载未完成时显示“CD封面加载中，请稍候……”，一旦加载完成，代理就把显示的职责委托给Icon。



设计CD封面虚拟代理

在开始写CD封面浏览器代码之前，让我们看一下类图。此类图和远程代理的图很类似，但是这里的代理是用于隐藏创建开销大的对象（因为我们需要通过网络取得图像数据），而不是隐藏在网络其他地方的对象。

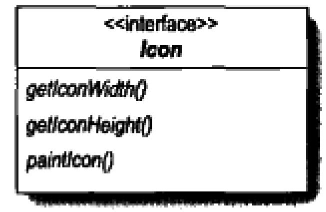


ImageProxy如何工作：

1. ImageProxy首先创建一个ImageIcon，然后开始从网络URL上加载图像。
2. 在加载的过程中，ImageProxy显示“CD封面加载中，请稍候……”。
3. 当图像加载完毕，ImageProxy把所有方法调用委托给真正的ImageIcon，这些方法包括了paintIcon()、getWidth()和getHeight()。
4. 如果用户请求新的图像，我们就创建新的代理，重复这样的过程。

编写ImageProxy

ImageProxy实现Icon接口。




```
class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;
                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(imageURL, "CD Cover");
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

此imageIcon是我们希望在加载后显示出来的真正的图像。

我们将图像的URL传入构造器中。这是我们希望显示的图像所在的位置。

在图像加载完毕前，返回默认的宽和高。图像加载完毕后，转给imageIcon处理。

有趣的地方在这里。这里的代码会在屏幕上画出一个icon图像（通过委托给imageIcon）。然而，如果我们没有被完整创建的ImageIcon，那就自己创建一个。下一页这一点你会看得更清楚……

再靠近一点

当需要在屏幕上绘制图像时，就调用此方法。

```
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
     if (imageIcon != null) {

         imageIcon.paintIcon(c, g, x, y);

     } else {

         g.drawString("Loading CD cover, please wait...", x+300, y+190);
         if (!retrieving) {

             retrieving = true;
             retrievalThread = new Thread(new Runnable() {
                 public void run() {
                     try {
                         imageIcon = new ImageIcon(imageURL, "CD Cover");
                         c.repaint();
                     } catch (Exception e) {
                         e.printStackTrace();
                     }
                 }
             });

             retrievalThread.start();
         }
     }
 }
```

如果你已经有icon，就告诉它画出自己。

否则，就显示那个"加载中"的消息。

我们在这里加载真正的icon图像。请注意，加载图像和ImageIcon是同步的（synchronous），也就是说，只有在加载完之后，ImageIcon构造器才会返回。这样，我们的程序会耗在这里，动弹不得，也没办法显示消息，所以要把加载变成异步的（asynchronous）。下一页会详细说明……

更靠近一点

如果我们还没有试着取出图像……

……那么就开始取出图像。（不要担心，只有一个线程会调用paint，所以这里的做法是线程安全的。）

我们不希望挂起整个用户界面，所以用另一个线程取出图像。

```
if (!retrieving) {
    retrieving = true;

    retrievalThread = new Thread(new Runnable() {
        public void run() {
            try {
                imageIcon = new ImageIcon(imageURL, "CD Cover");
                c.repaint();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
    retrievalThread.start();
}
```

在线程中，我们实例化此Icon对象，其构造器会在图像加载完成后才返回。

当图像准备好时，我们告诉Swing，需要重绘。

所以，下一次会在实例化ImageIcon之后，paintIcon方法才在屏幕上绘制真正的图像，而不是那个“加载中”的消息。

设计谜题

ImageProxy类似乎有两个，由条件语句控制的状态。你能否用另一个设计模式清理这样的代码？你要如何重新设计ImageProxy？

```
class ImageProxy implements Icon {
    // 实例变量构造器在这里
    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            // 这里有更多的代码
        }
    }
}
```

两个状态

两个状态

两个状态

测试CD封面浏览器



待烘烤代码

现在我们就来试试这个可爱的虚拟代理。我们已经烘烤好了一个新的ImageProxyTestDrive，用来设置窗口、创建框架、安装菜单和创建我们的代理。我们不在这里研究这些代码的细节，虚拟代理的代码列在本章最后，你可以随时去研究。

部分测试代码在下面：

```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{

        // 建立框架和菜单
        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

在这里我们创建一个图像代理，并指定初始URL。每次你从CD菜单中做出一个选择，就会得到一个新的图像代理。

接着，我们将代理包装进组件中，这样它就可以被放进框架。组件会处理代理的宽度、高度等细节。

最后我们把代理加进框架中，这样它可以被显示。

现在执行测试程序：

```
File Edit Window Help JustSomeOfTheCDsThatGotUsThroughThisBook
% java ImageProxyTestDrive
```

执行时，应该会看到这样的窗口。



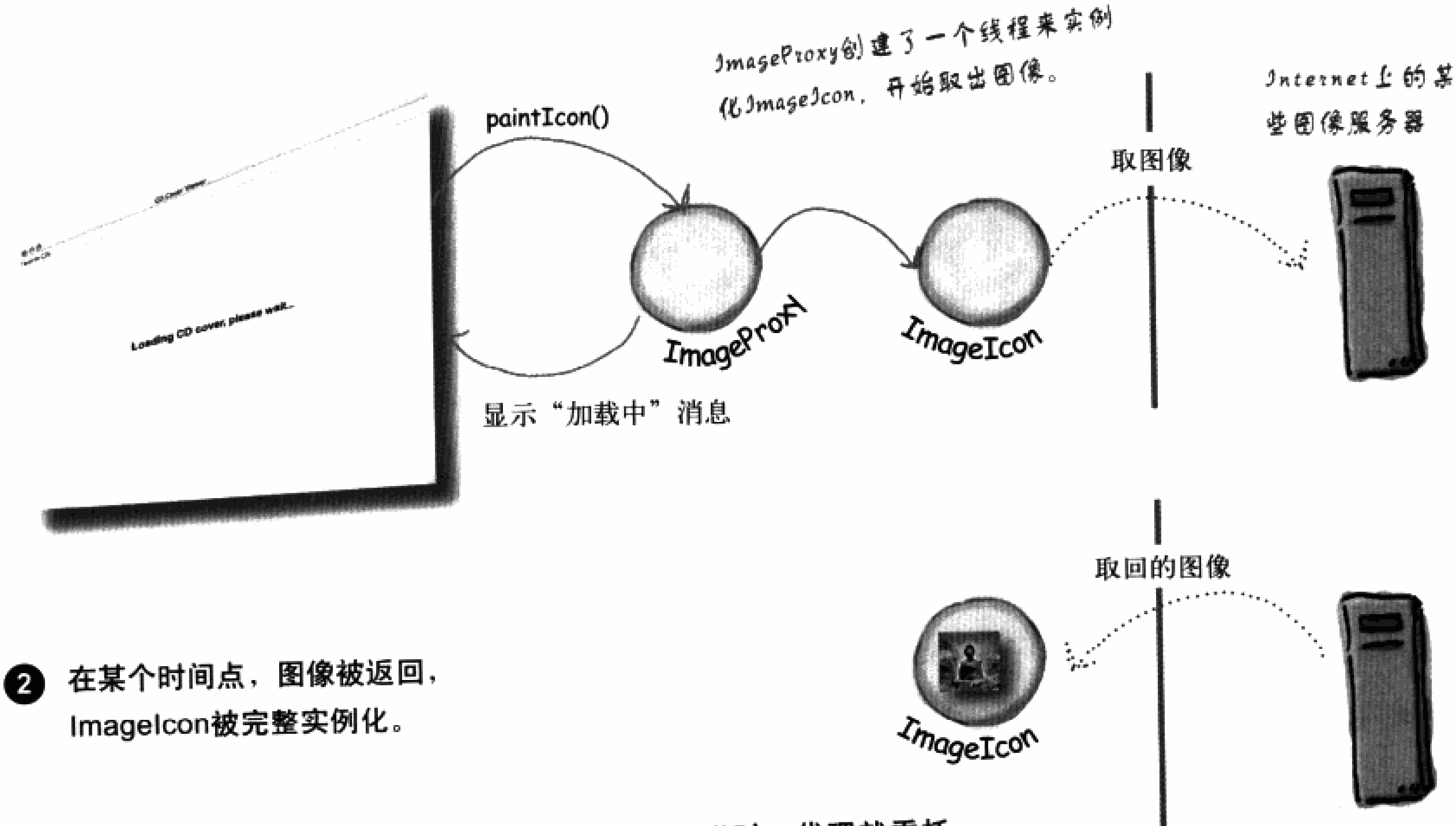
要测试的事情……

1. **用菜单加载不同的CD封面，然后看着代理显示“加载中”，直到出现真正的图像。**
2. **画面出现“加载中”消息时，缩放窗口大小，注意到代理会在不挂起Swing窗口的情况下处理加载。**
3. **在ImageProxyTestDrive中，加入一些你自己喜欢的CD。**

我们做了什么？

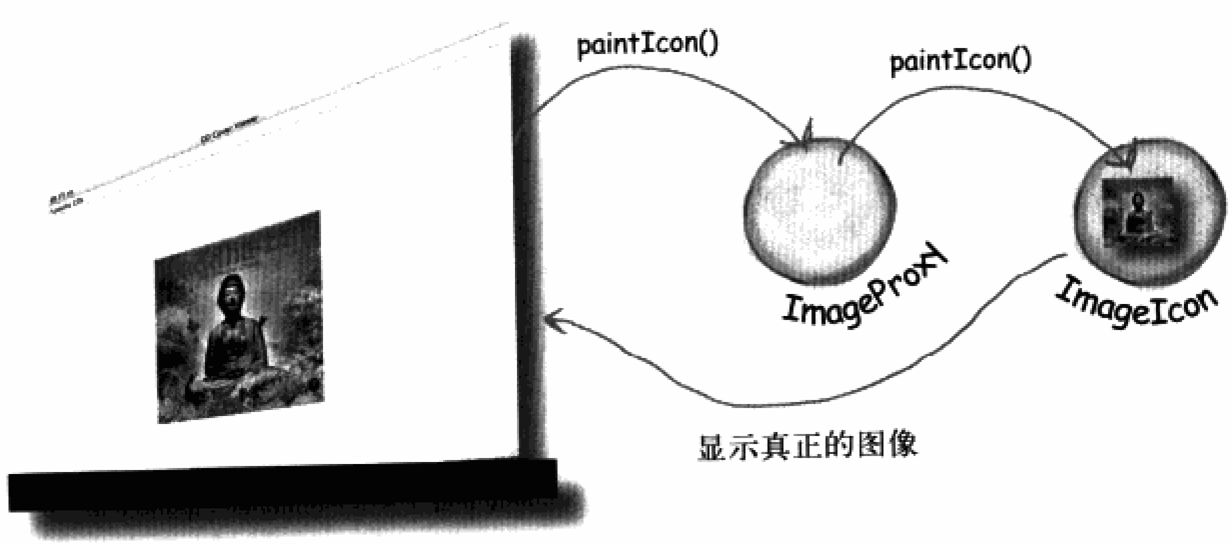


❶ 我们创建了一个用来显示的ImageProxy。paintIcon()方法会被调用，而ImageProxy会产生线程取得图像，并创建ImageIcon。



❷ 在某个时间点，图像被返回，ImageIcon被完整实例化。

❸ 在ImageIcon被创建后，下次调用到paintIcon()时，代理就委托ImageIcon进行。



there are no Dumb Questions

问：对我来说，远程服务器和虚拟服务器差异非常大，它们真的是一个模式吗？

答：在真实的世界中，代理模式有许多变体，这些变体都有共通点：都会将客户对主题（Subject）施加的方法调用拦截下来。这种间接的级别让我们可以做许多事，包括将请求分发到远程主题；给创建开销大的对象提供代表；或者正如你将要看到的，提供某些级别的保护，这种保护能决定哪些客户能调用哪些方法。这还只是个开端，其实一般的代理模式还可以以许多形式使用，本章最后我们会简略地提其中的几种变体。

问：ImageProxy在我看来好像是Decorator（装饰者）。我的意思是，我们基本上都是用一个对象把另一个包起来，然后把调用委托给ImageIcon。我这样说有什么问题吗？

答：有时候这两者的确看起来很像，但是它们的目的是不一样的。装饰者为对象增加行为，而代理是控制对象的访问。你可能会说："显示'加载中'消息，难道就不是在增加行为？"。从某方面来说，这的确可以算是，但是，更重要的，ImageProxy是控制ImageIcon的访问。如何控制呢？试想：代理将客户从ImageIcon解耦了，如果它们之间没有解耦，客户就必须等到每幅图像都被取回，然后才能把它绘制在界面上。代理控制ImageIcon的访问，以便在图像完全创建之前提供屏幕上的代表。一旦ImageIcon被创建，代理就允许访问ImageIcon。

问：我要如何让客户使用代理，而不是真正的对象？

答：好问题。一个常用的技巧是提供一个工厂，实例化并返回主题。因为这是在工厂方法内发生的，我们可以用代理包装主题再返回，而客户不知道也不在乎他使用的是代理还是真东西。

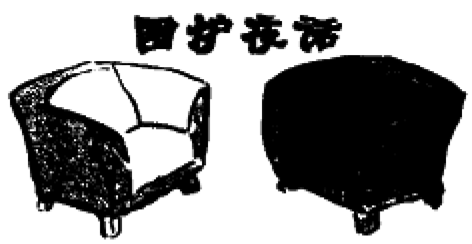
问：我注意到，在ImageProxy的例子中，你总是创建新的ImageProxy 来取得图像，即使图像已经被取回来过。能不能把加载过的图像放在缓存中呢？

答：你说的是缓存代理（Caching Proxy）。缓存代理会维护之前创建的对象，当收到请求时，在可能的情况下返回缓存对象。本章最后会介绍代理模式的几种变体。

问：我已经知道代理和装饰者的关系了，但是适配器呢？代理和适配器也很类似。

答：代理和适配器都是挡在其他对象的前面，并负责将请求转发给它们。适配器会改变对象适配的接口，而代理则实现相同的接口。

有一个额外相似性牵涉到保护代理（Protection Proxy）。保护代理可以根据客户的角色来决定是否允许客户访问特定的方法。所以保护代理可能只提供给客户部分接口，这就和某些适配器很相像了。再过几页，我们就会讨论到保护代理。



今夜话题：代理和装饰者的意图

代理

你好，装饰者。我猜你之所以会在这里，是因为人们常常把我们搞混了。

装饰者

我认为人们把我们搞混的原因是你到处招摇撞骗，说你是一个全然不同的模式。而事实上，你只不过是乔装过后的装饰者。我希望你不要这么喜欢抄袭我的想法。

代理

我抄袭你的想法？见鬼了！我控制对象的访问，你只是装饰对象，我的工作比你的重要多了。

装饰者

“只是”装饰对象？你认为装饰一点都不重要？我告诉你这个家伙，我为对象增加行为，这会改变对象的行为，你说重要不重要？

代理

好吧！或许你有那么一点意义……但是我还是不知道，你为什么认为我是在抄袭你。我是代表对象，不是装饰对象。

装饰者

你可以说这是“代表”，但如果看着像鸭子，走着像鸭子……我是说，看看你的虚拟代理吧！它只是加入行为的另一种方式，在创建开销大的对象时做一些事情，还有你的远程代理，就是一种和远程对象沟通的方法，这样客户就不用操心了。全都是关于行为，就像我所说的。

代理

装饰者，我想你还是没搞懂。我代表对象，不光是为对象加上动作。客户使用我作为真正主题的替身，因为我可以保护对象避免不想要的访问，也可以避免在加载大对象的过程中GUI会挂起，或者隐藏主题在远程运行的事实。我的意图和你的差别很大！

装饰者

你爱怎么说都可以。我实现和所包装对象相同的接口，你不也是吗！

代理

好，听听你说了什么。你说你包装了一个对象。有时候我们非正式地说：代理包装了它的主题，这样说其实并不准确。

想想远程代理……我包装了什么对象？我所代表和控制访问的对象是在另一台机器上呀！你能办得到吗？

当然有了，以虚拟代理来说……想想CD浏览器的例子。当客户第一次用我当做代理的时候，主题甚至还根本不存在呢！你说这次我又包装谁了？

我不知道你这么笨！当然我有时候会创建对象，不然你以为虚拟代理是怎么取得主题的！好了，你刚刚指出了我们之间的一个大差异：我们都知道装饰者只能装饰点缀，你们从来不会实例化任何东西。

嘿，经过这次谈话，我确信你是个笨蛋代理。

你很少看到代理将一个主题包装多次，事实上，如果你真的把某些对象包装十次，你最好回去重新检查你的设计。

装饰者

怎么说？

好吧！但远程代理毕竟是特例，我不相信你可以找出另一个例子来。

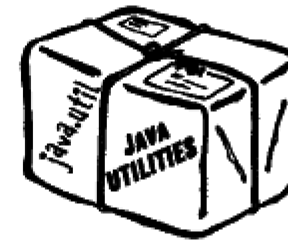
哼哼，我猜接下来你甚至会说对象其实是你创建的。

是吗？实例化这个吧！（做了个令人作呕的动作。）

你说我笨蛋？我倒想看看你有没有能耐将一个对象包装十层，手还不会酸。

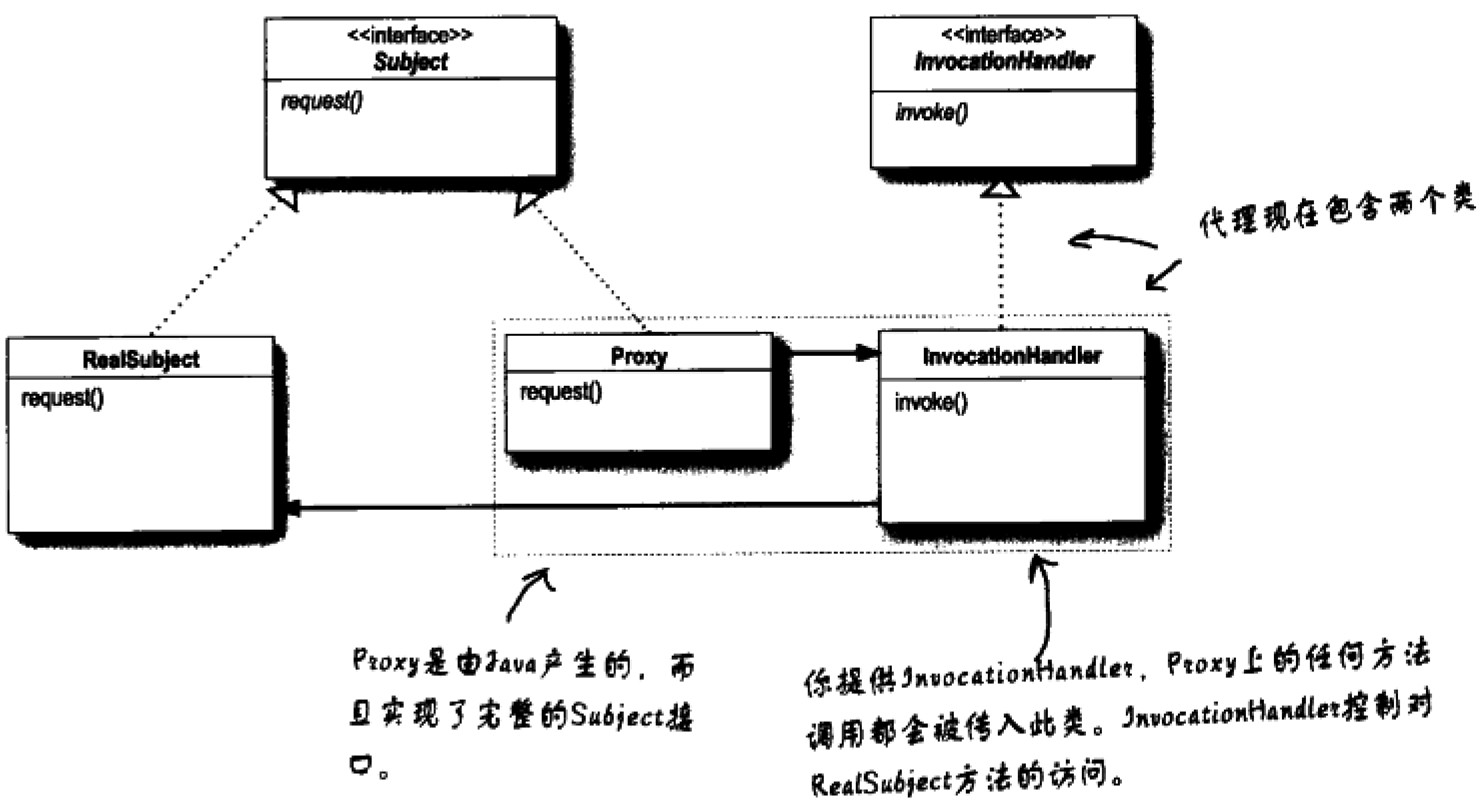
你们代理就是这样，装腔作势的功夫是一流的，好像你们有真本事。我真替你感到可怜。

使用Java API的代理，创建一个保护代理



Java在java.lang.reflect包中有自己的代理支持，利用这个包你可以在运行时动态地创建一个代理类，实现一个或多个接口，并将方法的调用转发到你所指定的类。因为实际的代理类是在运行时创建的，我们称这个Java技术为：动态代理。

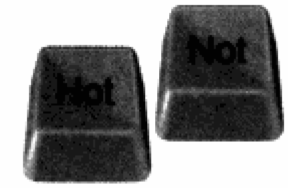
我们要利用Java的动态代理创建我们下一个代理实现（保护代理）。但在这之前，先让我们看一下类图，了解一下动态代理是怎么一回事。就和真实世界中大多数的事物一样，它和代理模式的传统定义有一点出入。



因为Java已经为你创建了Proxy类，所以你需要有办法来告诉Proxy类你要做什么。你不能像以前一样把代码放在Proxy类中，因为Proxy不是你直接实现的。既然这样的代码不能放在Proxy类中，那么要放在哪里？放在InvocationHandler中。InvocationHandler的工作是响应代理的任何调用。你可以把InvocationHandler想成是代理收到方法调用后，请求做实际工作的对象。

接下来，看看如何使用动态代理……

对象村的配对



每个城镇都需要配对服务，不是吗？你负责帮对象村实现约会服务系统。你有一个好点子，就是在服务中加入“Hot”和“Not”的评鉴，“Hot”就表示喜欢对方，“Not”表示不喜欢。你希望这套系统能鼓励你的顾客找到可能的配对对象，这也会让事情更有趣。

你的服务系统涉及到一个Person bean，允许设置或取得一个人的信息：

这是一个接口，我们稍后就会实现它……

这里我们可以取得人的名字、性别、兴趣和HotOrNot评分（1到10）。

```
public interface PersonBean {

    String getName();
    String getGender();
    String getInterests();
    int getHotOrNotRating();

    void setName(String name);
    void setGender(String gender);
    void setInterests(String interests);
    void setHotOrNotRating(int rating);

}
```

通过调用各自的方法，我们也可以设置这些信息。

setHotOrNotRating()需要一个整数作为参数，并将它加入此人的运行平均值中。

现在，让我们看看实现……

PersonBean的实现

PersonBeanImpl实现了PersonBean接口。

```
public class PersonBeanImpl implements PersonBean {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getHotOrNotRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }


    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setHotOrNotRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

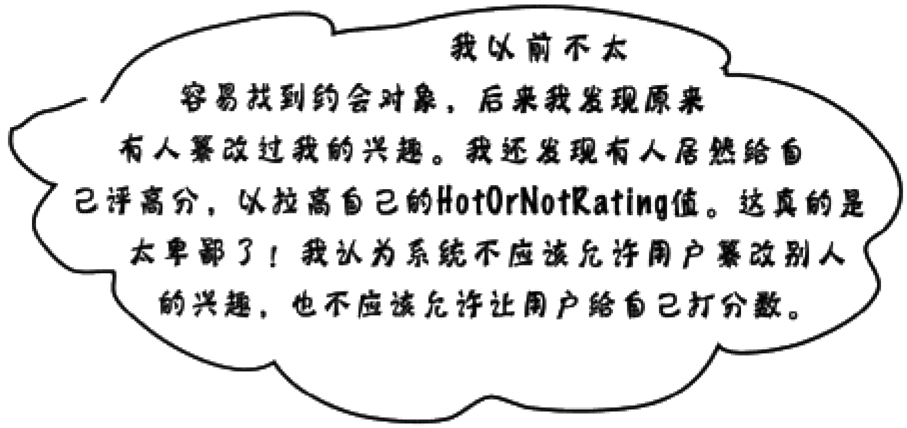
实例变量。

这些Getter方法各自返回相应的实例变量……

除了getHotOrNotRating()，它计算rating的平均值（rating/ratingCount）。

……这是所有的Setter方法，设定相应的实例变量。

最后，setHotOrNotRating()增加计数值，并将参数加到rating实例变量中。





虽然我们怀疑Elroy找不到约会对象可能是因为其他的因素，但是他说的没错，系统不应该允许用户篡改别人的数据。根据我们定义PersonBean的方式，任何客户都可以调用任何方法。

这是一个我们可以使用保护代理的绝佳例子。什么是保护代理？这是一种根据访问权限决定客户可否访问对象的代理。比方说，如果你有一个雇员对象，保护代理允许雇员调用对象上的某些方法，经理还可以多调用一些其他的方法（像setSalary()），而人力资源处的雇员可以调用对象上的所有方法。

在我们的约会服务中，我们希望顾客可以设置自己的信息，同时又防止他人更改这些信息。HotOrNot评分则相反，你不能更改自己的评分，但是他人可以设置你的评分。我们在PersonBean中已经有许多getter方法了，每个方法的返回信息都是公开的，任何顾客都可以调用它们。

五分钟短剧：保护主题

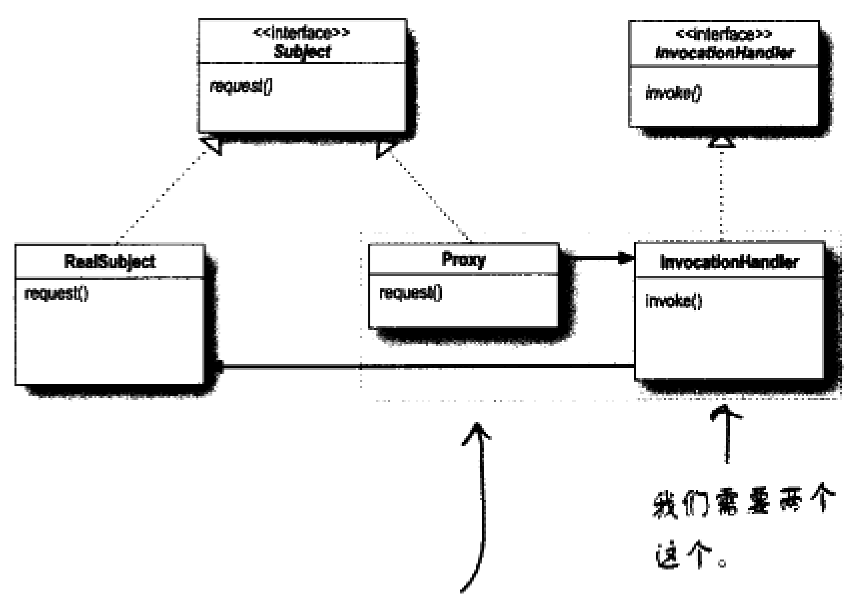
Internet的泡沫已经渐渐被人们淡忘了。在那些日子里，如果你需要找一个更好更高薪的工作，对街就找得到。甚至软件开发人员的经纪人也赶上这股风潮……



大局观：为PersonBean创建动态代理

我们有一些问题要修正：顾客不可以改变自己的HotOrNot评分，也不可以改变其他顾客的个人信息。要修正这些问题，你必须创建两个代理：一个访问你自己的PersonBean对象，另一个访问另一顾客的PersonBean对象。这样，代理就可以控制在每一种情况下允许哪一种请求。

创建这种代理，我们必须使用Java API的动态代理，在几页前有这个API的概况。Java会为我们创建两个代理，我们只需要提供handler来处理代理转来的方法。



步骤一：

创建两个InvocationHandler。

InvocationHandler实现了代理的行为，正如你将看到的，Java负责创建真实代理类和对象。我们只需提供在方法调用发生时知道做什么的handler。

步骤二：

写代码创建动态代理。

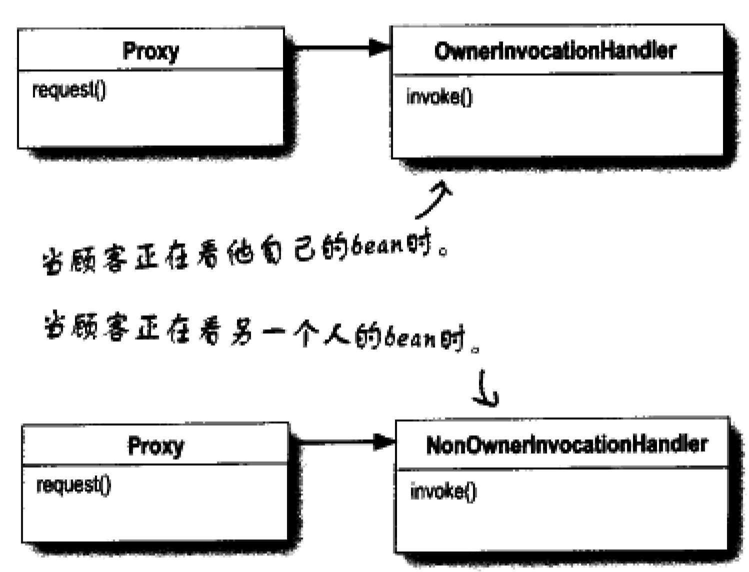
我们需要写一些代码产生代理类，并实例化它。等一下你就会看到这些代码。

步骤三：

利用适当的代理包装任何PersonBean对象。

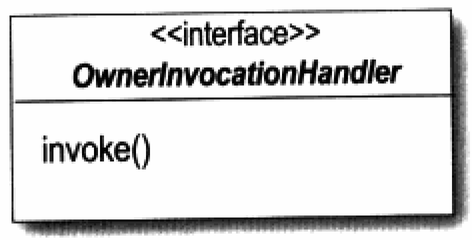
当我们需要使用PersonBean对象时，如果不是顾客自己（在这种情况下，称为“拥有者”），就是另一个顾客正在检查的服务使用者（在这种情况下，我们叫它“非拥有者”）。

不管是哪一种情况，我们都为PersonBean创建适合的代理。



步骤一：创建InvocationHandler

我们知道需要写两个InvocationHandler（调用处理器），其中一个给拥有者使用，另一个给非拥有者使用。究竟什么是InvocationHandler呢？你可以这么想：当代理的方法被调用时，代理就会把这个调用转发给InvocationHandler，但是这并不是通过调用InvocationHandler的相应方法做到的。那么，是如何做到的？让我们看看InvocationHandler的接口：



这里只有一个名为invoke()的方法，不管代理被调用的是何种方法，处理器被调用的一定是invoke()方法。让我们看看这是如何工作的：

① 假设proxy的setHotOrNotRating()方法被调用。

```
proxy.setHotOrNotRating(9);
```

② proxy会接着调用InvocationHandler的invoke()方法。

```
invoke(Object proxy, Method method, Object[] args)
```

Method类是reflection API的一部分，利用它的getName()方法，我们就可以知道proxy被调用的方法是什么。

③ handler决定要如何处置这个请求，可能会转发给RealSubject。handler到底是如何决定的呢？等一下你就知道了。

调用RealSubject方法的代码。

```
return method.invoke(person, args);
```

我们调用原始proxy被调用的方法。这个对象在调用时被传给我们。只不过加载调用的是真正的主题（person）。

这是被调用的对象。

使用原始的变量。

继续创建InvocationHandler……

当proxy调用invoke()时，要如何应对？通常，你会先检查该方法是否来自proxy，并基于该方法的名称和变量做决定。现在我们就来实现OwnerInvocationHandler，以了解工作机制：

InvocationHandler是java.lang.reflect包的一部分，所以我们需要import它。

所有调用处理器都实现InvocationHandler接口。

我们将person传入构造器，并保持它的引用。

每次proxy的方法被调用，就会导致proxy调用此方法。

如果方法是一个getter，我们就调用person内的方法。

否则，如果方法是setHotOrNotRating()，我们就抛出IllegalAccessException表示不允许。

如果真正主题抛出异常的话，就会执行这里。

因为我们是拥有者，所以任何其他set方法都可以，我们就在真正主题上调用它。

如果调用其他的方法，一律不理，返回null。

```
import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public OwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

练习

NonOwnerInvocationHandler 工作的方式除了它允许调用setHotOrNotRating()和不允许调用其他set方法之外，与OwnerInvocationHandler是很相似的。请写出NonOwnerInvocationHandler的代码：

步骤二：创建Proxy类并实例化Proxy对象

现在，只剩下创建动态Proxy类，并实例化Proxy对象了。让我们开始编写一个以PersonBean为参数，并知道如何为PersonBean对象创建拥有者代理的方法。也就是说，我们要创建一个代理，将它的方法调用转发给OwnerInvocationHandler。代码如下：

此方法需要一个person对象作为参数，然后返回它的代理，因为代理和主题有相同的接口，所以我们返回一个PersonBean。

此代码创建了代理，这个代码有点丑，所以要小心查看它。

我们利用Proxy类的静态newProxyInstance方法创建代理……

```
PersonBean getOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new OwnerInvocationHandler(person));
}
```

← 将personBean的类载入器当做参数……

……代理需要实现的接口……

将person传入调用处理器的构造器中。如果你回到几页前，就会发现这正是处理器能够访问真实主题的原因。

……调用处理器（在这里是OwnerInvocationHandler）。

Sharpen your pencil

虽然有一点复杂，但是创建动态代理所需要的代码其实很短。请你写下getNonOwnerProxy()，该方法会返回NonOwnerInvocationHandler的代理：

更进一步：你能够写下getProxy()方法，参数是handler和person，返回值是使用此handler的代理吗？

测试配对服务

现在我们就来试试配对服务，看看代理如何控制对setter方法的访问。

```
public class MatchMakingTestDrive {
    // 这里有实例变量

    public static void main(String[] args) {
        MatchMakingTestDrive test = new MatchMakingTestDrive();
        test.drive();
    }

    public MatchMakingTestDrive() {
        initializeDatabase();
    }
    public void drive() {
        PersonBean joe = getPersonFromDatabase("Joe Javabean");
        PersonBean ownerProxy = getOwnerProxy(joe);
        System.out.println("Name is " + ownerProxy.getName());
        ownerProxy.setInterests("bowling, Go");
        System.out.println("Interests set from owner proxy");
        try {
            ownerProxy.setHotOrNotRating(10);
        } catch (Exception e) {
            System.out.println("Can't set rating from owner proxy");
        }
        System.out.println("Rating is " + ownerProxy.getHotOrNotRating());

        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Name is " + nonOwnerProxy.getName());
        try {
            nonOwnerProxy.setInterests("bowling, Go");
        } catch (Exception e) {
            System.out.println("Can't set interests from non owner proxy");
        }
        nonOwnerProxy.setHotOrNotRating(3);
        System.out.println("Rating set from non owner proxy");
        System.out.println("Rating is " + nonOwnerProxy.getHotOrNotRating());
    }

    // 这里还有其他的方法，像getOwnerProxy和getNonOwnerProxy
}
```

main()创建测试程序对象，调用其drive()方法开始测试。

构造器初始化配对服务人员数据库。

从数据库中取出一个人。

……然后创建一个拥有者代理。

调用getter。

然后调用setter。

试着改变评分。

这应该是行不通的！

创建一个非拥有者代理。

……调用getter。

跟着调用setter。

这应该是行不通的！

试着设置评分。

这应该行得通！

执行结果

```
File Edit Window Help Born2BDynamic
% java MatchMakingTestDrive
Name is Joe Javabean
Interests set from owner proxy
Can't set rating from owner proxy
Rating is 7

Name is Joe Javabean
Can't set interests from non owner proxy
Rating set from non owner proxy
Rating is 5
%
```

我们的OwnerProxy允许getter和setter，但不允许改变HotOrNot评分。

我们的NonOwnerProxy只允许getter和改变HotOrNot评分，但不允许setter。

there are no Dumb Questions

问：到底"动态代理"动态在哪里？是不是指在运行时才将它实例化并和handler联系起来？

答：不是的。动态代理之所以被称为动态，是因为运行时才将它的类创建出来。代码开始执行时，还没有proxy类，它是根据需要从你传入的接口集创建的。

问：我的InvocationHandler看起来像一个很奇怪的proxy。它没有实现所代理的类的任何方法。

答：这是因为Invocation-Handler根本就不是proxy，它只是一个帮助proxy的类，proxy会把调用转发给它处理。Proxy本身是利用静态的Proxy.newProxyInstance()方法在运行时动态地创建的。

问：有没有办法知道某个类是不是代理类呢？

答：可以。代理类有一个静态方法，叫做isProxyClass()。此方法的返回值如果为true，表示这是一个动态代理类。除此之外，代理类还会实现特定的某些接口。

问：对于我能传入new-ProxyInstance()的接口类型，有没有什么限制？

答：是有一些限制。首先，我们总是传给newProxyInstance()一个接口数组，此数组内只能有接口，不能有类。如果接口不是public，就必须属于同一个package，不同的接口内，不可以有名称和参数完全一样的方法。还有一些比较细微的限制，你应该好好研读一下javadoc的文件。

问：你为什么使用skeleton？我以为我们早在Java 1.2就已经摆脱skeleton了。

答：你说的没错，我们不需要真的产生skeleton，因为Java 1.2的RMI可以利用reflectionAPI直接将客户调用分派给远程服务。尽管如此，我们还是希望呈现skeleton，因为这可以帮助你从概念上理解内部的机制。

问：我听说，在Java 5，甚至连stub都不需要产生了，这是真的吗？

答：是真的。Java 5的RMI和动态代理搭配使用，动态代理动态产生stub，远程对象的stub是java.lang.reflect.Proxy实例（连同一个调用处理器），它是自动产生的，来处理所有把客户的本地调用变成远程调用的细节。所以，你不再需要使用rmic，客户和远程对象沟通的一切都在幕后处理掉了。

连连看

请将下列模式和描述配对:

模式	描述
装饰者	包装另一个对象，并提供不同的接口。
外观	包装另一个对象，并提供额外的行为。
代理	包装另一个对象，并控制对它的访问。
适配器	包装许多对象以简化它们的接口。

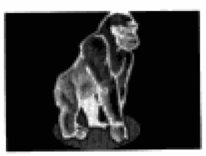
代理动物园



欢迎来到对象村动物园！

现在你知道什么是远程代理、虚拟代理和保护代理了。在野外，你看到的代理还不只是这些。在动物园的代理区，我们展示了许多辛苦捕捉来的野生的代理，供你研究。

我们的工作还没有完成，但是，我们相信以后你会在真实世界中看到更多代理的变体，所以现在请你帮帮忙，帮我们编目。让我们看看现有的代理：



防火墙代理（Firewall Proxy）

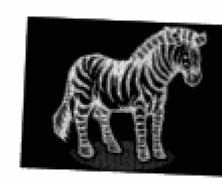
控制网络资源的访问，保护主题免于“坏客户”的侵害。

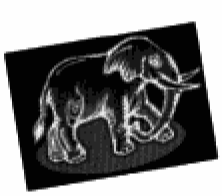
栖息地：常出没于公司的防火墙系统。

帮助找到栖息地

智能引用代理（Smart Reference Proxy）

当主题被引用时，进行额外的动作，例如计算一个对象被引用的次数。





缓存代理（Caching Proxy）

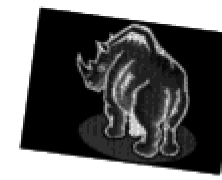
为开销大的运算结果提供暂时存储：它也允许多个客户共享结果，以减少计算或网络延迟。

栖息地：常出没于Web服务器代理，以及内容管理与出版系统。



被发现出没于JavaSpaces，为分散式环境内的潜在对象集合提供同步访问控制。

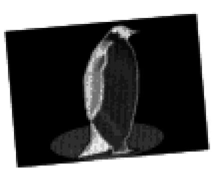
同步代理（Synchronization Proxy）在多线程的情况下为主题提供安全的访问。



复杂隐藏代理（Complexity Hiding Proxy）

用来隐藏一个类的复杂集合的复杂度，并进行访问控制。有时候也称为外观代理（Façade Proxy），这不难理解。复杂隐藏代理和外观模式是不一样的，因为代理控制访问，而外观模式只提供另一组接口。

帮忙找出栖息地

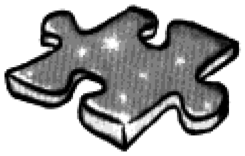


写入时复制代理（Copy-On-Write Proxy）

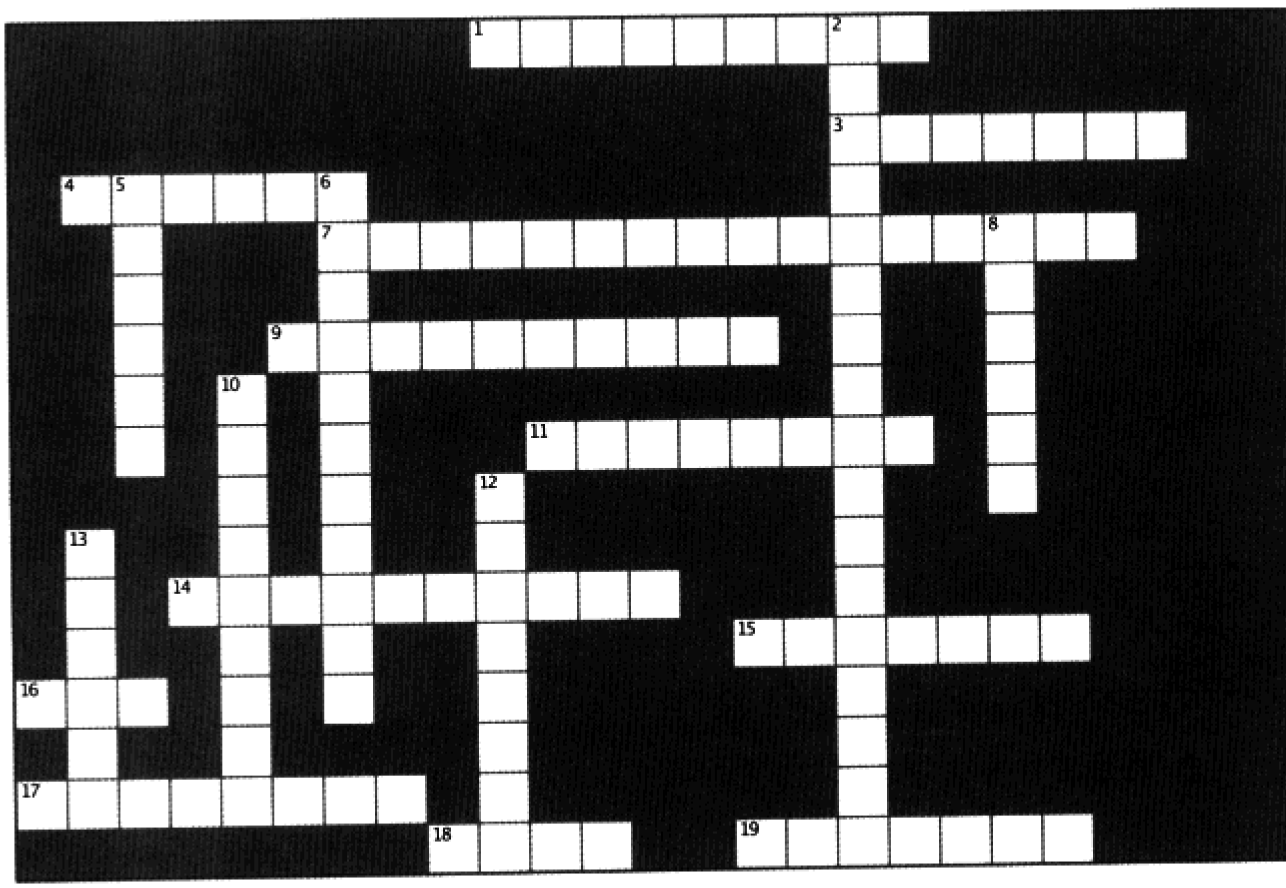
用来控制对象的复制，方法是延迟对象的复制，直到客户真的需要为止。这是虚拟代理的变体。

栖息地：去看看Java 5的CopyOnWriteArrayList附近。

注意：请将你在野外所观察到的其他代理写在这里：



这一章很长。在结束前，休闲一下吧！

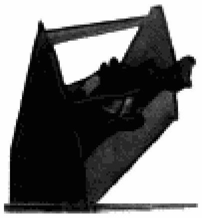


横排提示：

1. Group of first CD cover displayed (two words)
3. Proxy that stands in for expensive objects
4. We took one of these to learn RMI
7. Remote ________ was used to implement the gumball machine monitor (two words)
9. Software developer agent was being this kind of proxy
11. In RMI, the object that takes the network requests on the service side
14. Proxy that protects method calls from unauthorized callers
15. A ________ proxy class is created at runtime
16. Place to learn about the many proxy variants
17. Commonly used proxy for web services (two words)
18. In RMI, the proxy is called this
19. The CD viewer used this kind of proxy

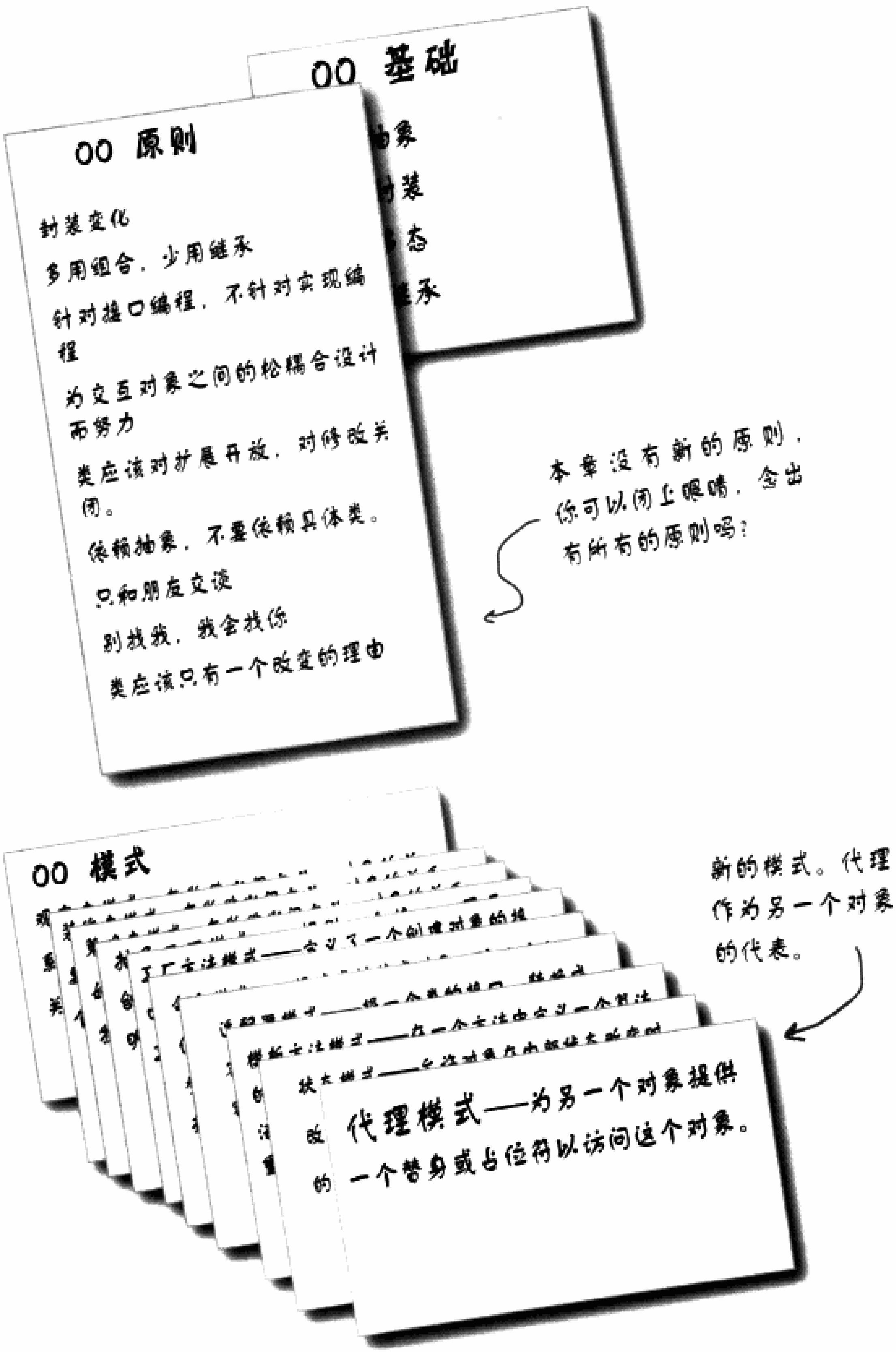
竖排提示：

2. Java's dynamic proxy forwards all requests to this (two words)
5. Group that did the album MCMXC A.D.
6. This utility acts as a lookup service for RMI
8. Why Elroy couldn't get dates
10. Similar to proxy, but with a different purpose
12. Objectville Matchmaking gimmick (three words)
13. Our first mistake: the gumball machine reporting was not ________



设计箱内的工具

你的设计工具箱几乎满了。一路下来，你所学会的设计模式，几乎可以解决任何设计问题了。



要点

- 代理模式为另一个对象提供代表，以便控制客户对对象的访问，管理访问的方式有许多种。
- 远程代理管理客户和远程对象之间的交互。
- 虚拟代理控制访问实例化开销大的对象。
- 保护代理基于调用者控制对对象方法的访问。
- 代理模式有许多变体，例如：缓存代理、同步代理、防火墙代理和写入时复制代理。
- 代理在结构上类似装饰者，但是目的不同。
- 装饰者模式为对象加上行为，而代理则是控制访问。
- Java内置的代理支持，可以根据需要建立动态代理，并将所有调用分配到所选的处理器。
- 就和其他的包装者（wrapper）一样，代理会造成你的设计中类的数目增加。

习题解答

练习

NonOwnerInvocationHandler工作的方式除了它允许调用setHotOrNotRating()和不允许调用其他set方法之外，与OwnerInvocationHandler是很相似的。请写出NonOwnerInvocationHandler的代码：

```
import java.lang.reflect.*;

public class NonOwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public NonOwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

设计迷题

ImageProxy类似乎有两个由条件语句控制的状态。你能否用另一个设计模式清理这样的代码？你要如何重新设计ImageProxy？

使用状态模式：实现两个状态，分别是ImageLoaded和ImageNotLoaded。然后把if语句内的代码放进各自的状态中。一开始的状态是ImageNotLoaded，当ImageIcon取回后就转换到ImageLoaded状态。

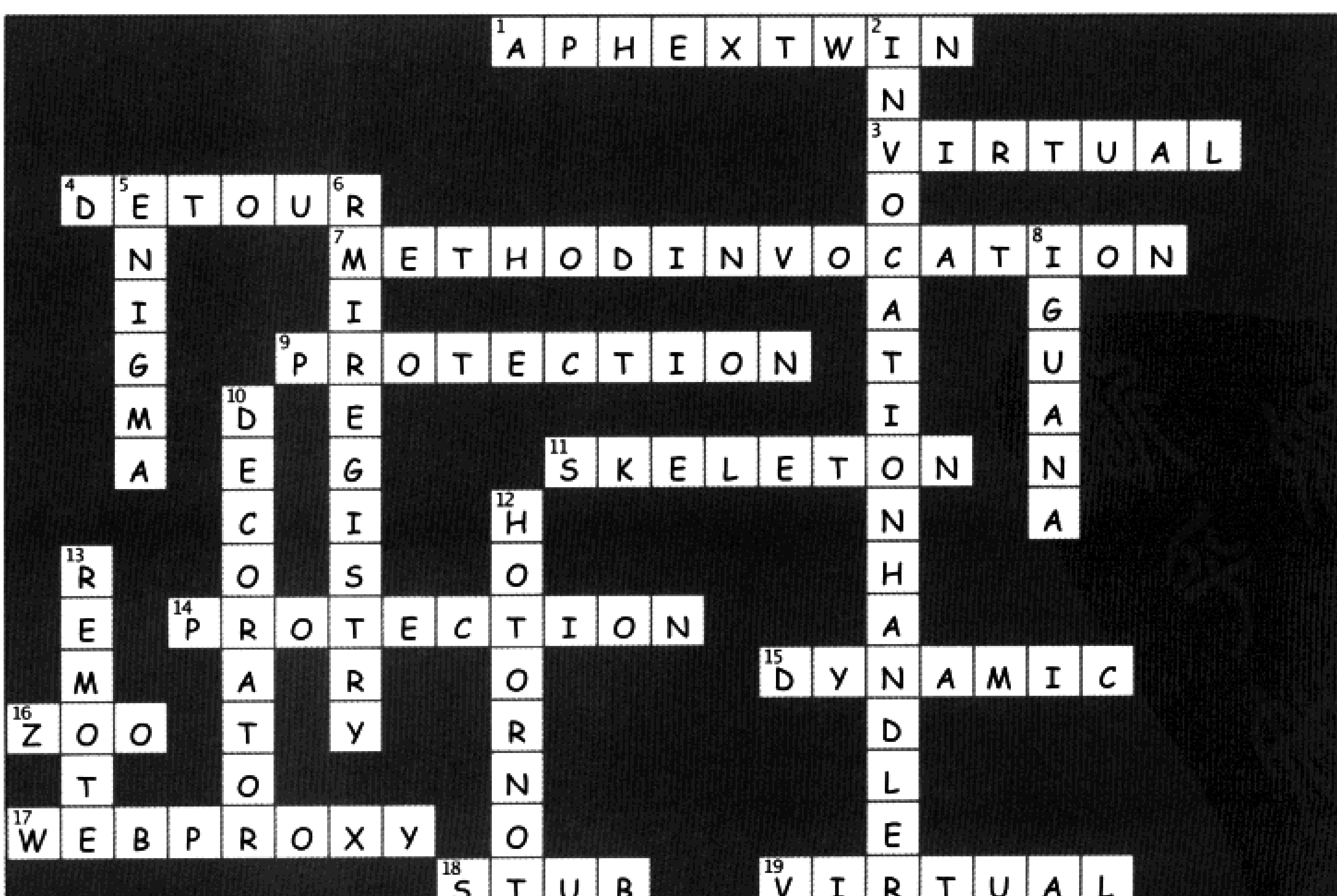
习题解答

Sharpen your pencil

虽然有一点复杂，但是创建动态代理所需要的代码其实很短。请你写下getNonOwnerProxy()，该方法会返回NonOwnerInvocationHandler的代理：

```
PersonBean getNonOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new NonOwnerInvocationHandler(person));
}
```





待烘烤
代码

CD封面浏览器的代码

```
package headfirst.proxy.virtualproxy;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("CD Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable cds = new Hashtable();

    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

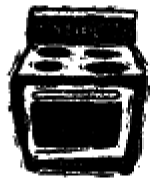
    public ImageProxyTestDrive() throws Exception{
        cds.put("Ambient: Music for Airports","http://images.amazon.com/images/P/
B000003S2K.01.LZZZZZZZ.jpg");
        cds.put("Buddha Bar","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.
jpg");
        cds.put("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg");
        cds.put("Karma","http://images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.gif");
        cds.put("MCMXC A.D.","http://images.amazon.com/images/P/B000002URV.01.LZZZZZZZ.
jpg");
        cds.put("Northern Exposure","http://images.amazon.com/images/P/B000003SFN.01.
LZZZZZZZ.jpg");
        cds.put("Selected Ambient Works, Vol. 2","http://images.amazon.com/images/P/
B000002MNZ.01.LZZZZZZZ.jpg");
        cds.put("oliver","http://www.cs.yale.edu/homes/freeman-elisabeth/2004/9/Oliver_
sm.jpg");

        URL initialURL = new URL((String)cds.get("Selected Ambient Works, Vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite CDs");
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);
```

```
        for(Enumeration e = cds.keys(); e.hasMoreElements();) {
            String name = (String)e.nextElement();
            JMenuItem menuItem = new JMenuItem(name);
            menu.add(menuItem);
            menuItem.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent event) {
                     imageComponent.setIcon(new ImageProxy(getCDUrl(event.
getActionCommand())));
                    frame.repaint();
                  }
            });
        }

        // 建立框架和菜单
        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }
    URL getCDUrl(String name) {
        try {
            return new URL((String)cds.get(name));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```



待烘烤代码

CD封面浏览器的代码，继续……

```
package headfirst.proxy.virtualproxy;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;

                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(imageURL, "CD Cover");
                            c.repaint();
                        } catch (Exception e) {
```

```
                        e.printStackTrace();
                    }
                }
            });
            retrievalThread.start();
        }
    }
  }
}
```

```
package headfirst.proxy.virtualproxy;
import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icon.getIconWidth();
        int h = icon.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icon.paintIcon(this, g, x, y);
    }
}
```

12 复合模式

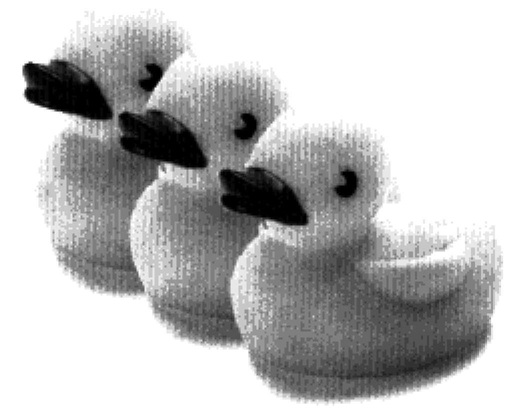
模式的模式



谁料得到模式居然可以携手合作？ 你已经见识过围炉夜话的火爆场面（幸好，出版社事先请我们删除“死神来访”模式的篇章，好让本书不需附上“12岁以下读者必须家长陪同阅读”的警告标语，所以你没见识到闹出人命的那一集围炉夜话★），谁料得到模式居然可以携手合作？这实在是太意外了。信不信由你，有一些威力强大的OO设计同时使用多个设计模式。准备让你的模式技巧进入下一个层次，现在是复合模式的时间。

★如果你想要一份，来E-mail索取。

携手合作



使用模式最棒的方式，就是把它们从家里找出来同其他模式展开交互。你越多地使用模式就越容易发现它们一同现身在你的设计中。对于这些在设计中携手合作征服许多问题的模式，我们给它一个特别的名字：复合模式（Compound Pattern）。没错！我们说的正是一种由模式所构成的模式。

你将在真实的世界中发现许多复合模式。现在你的大脑中已经有许多模式了，对于复合模式，你会发现它们其实只是携手合作的许多模式，这样就会很容易理解了。

本章，我们将重访SimUDuck鸭子模拟器中那些熟悉的鸭子。当我们介绍复合模式时，使用鸭子的例子是适当的，毕竟，在整本书中，鸭子一直与我们同在，而且模拟鸭子也使用了许多模式。通过鸭子的帮助，你将学习到模式如何携手合作来解决同一件事。但是我们将某些模式结合使用，并不代表这些模式就够资格称为复合模式。复合模式必须够一般性，适合解决许多问题才行。因此，在本章的后半段，我们会拜访一个真正的复合模式，没错，就是鼎鼎大名的MVC（Model-View-Controller）。如果你没听过MVC，我保证这会是你的设计工具箱内最有威力的模式之一。

模式通常被一起使用，并被组合在同一个设计解决方案中。

复合模式在一个解决方案中结合两个或多个模式，以解决一般或重复发生的问题。

与鸭子重聚

正如你所知道的，我们会再度与鸭子共同合作。而这次鸭子将在同一个解决方案中展示模式是如何共存甚至携手合作的。

我们将从头重建我们的鸭子模拟器，并通过使用一堆模式来赋予它一些有趣的能力。动工了……

① **首先，我们将创建一个Quackable接口。**

刚刚说过，我们将从头开始。而这一次，鸭子将实现Quackable接口。这样，我们就知道这个模拟器中，有哪些东西可以呱呱叫，像是绿头鸭、红头鸭，甚至可能还会看到橡皮鸭偷偷溜回来。

```
public interface Quackable {
    public void quack();
}
```

Quackable只需做好一件事：Quack（呱呱叫）！

② **现在，某些鸭子实现了Quackable接口。**

如果没有类实现某个接口，那么此接口的存在就没有意义。现在我们就来设计一些具体鸭子（不是那种“玩偶鸭”，你知道我们指的是什么）。

标准的绿头鸭。

```
public class MallardDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

```
public class RedheadDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

如果我们希望这个模拟器活泼有趣，就要一些物种变体。

如果我们没有加入了别的种类的鸭子，就不太好玩。

还记得上次吧？我们曾经加入了鸭鸣器（猎人使用的那种东西，它们肯定会呱呱叫）和橡皮鸭。

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
```

DuckCall（鸭鸣器）会呱呱叫，但听起来并不十分像真的鸭叫声。

```
public class RubberDuck implements Quackable {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

当RubberDuck（橡皮鸭）呱呱叫时，其实声音是吱吱叫。

③ 好了，我们有了鸭子，还需要一个模拟器。

让我们来制造一个会产生一些鸭子，还要确认鸭子会呱呱叫的模拟器……

我们的main()方法将让所有的事情动起来。

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

我们创建一个模拟器，然后调用其simulate()方法。

我们需要一些鸭子，所以在这里把每一种会呱呱叫的东西都产生一份实例……

……然后开始模拟每种鸭子。

我们在这里重载了simulate()方法来模拟一只鸭子。

剩下的事，我们就让多态发挥它的魔法：不管传入的是哪一种呱呱叫对象，多态都可以调用到正确的方法。

不要太兴奋，我们还没加上模式呢！

```
% java DuckSimulator

Duck Simulator
Quack
Quack
Kwak
Squeak

%
```

大家都实现同一个Quackable接口，只是各自的实现允许不同的呱呱叫方式。

似乎到目前为止一切顺利。

④ 当鸭子出现在这里时，鹅也应该在附近。

只要有水塘的地方，就大概会有鸭子和鹅。我们为这个模拟器设计了一个Goose（鹅）类。

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

鹅的叫声是咯咯，而不是呱呱。

BRAIN POWER

假设我们想要在所有使用鸭子的地方使用鹅，毕竟鹅会叫、会飞、会游，和鸭子差不多。为什么我们不能在这个模拟器中使用鹅呢？

什么模式可以让我们轻易地将鸭子和鹅掺杂在一起呢？

⑤ 我们需要鹅适配器

我们的模拟器期望看到Quackable接口。既然鹅不会呱呱叫，那么我们可以利用适配器将鹅适配成鸭子。

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }
}
```

请牢记，适配器会实现目标接口，也就是Quackable。

构造器需要传入要适配的鹅对象。

当调用quack()时，会被委托到鹅的honk()方法。

⑥ 现在，模拟器中也应该可以使用鹅了。

接着，我们需要做的就是创建Goose对象，将它包装进适配器，以便实现Quackable。这样，我们就可以继续了。

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

通过把Goose包装进GooseAdapter，我们就可以让鹅像鸭子一样。

一旦鹅被包装起来，我们就可以把它当做其他鸭子的Quackable对象。

⑦ 现在，让我们测试看看……

这次测试时，simulate()会调用许多对象的quack()方法，其中包括适配器的quack()方法。结果应该会出现咯咯叫（honk）才对。

```
File Edit Window Help GoldenEggs
% java DuckSimulator

Duck Simulator: With Goose Adapter
Quack
Quack
Kwak
Squeak
Honk

%
```

有鹅了！现在鹅会和其他鸭子一起叫出声。



呱呱叫学

呱呱叫学家为所有拥有可呱呱叫行为的事物着迷。其中一件他们经常研究的事是：在一群鸭子中，会有多少呱呱叫声？

我们要如何在不变化鸭子类的情况下，计算呱呱叫的次数呢？

有没有什么模式可以帮上忙？

J. Brewer，公园巡游者和呱呱叫学家



⑧ 我们会让这些呱呱叫学家满意，让他们知道叫声的次数。

怎样才能办到呢？让我们创建一个装饰者，通过把鸭子包装进装饰者对象，给鸭子一些新行为（计算次数的行为）。我们不必修改鸭子的代码。

QuackCounter是一个装饰者。

像适配器一样，我们需要实现目标接口。

我们用一个实例变量来记录被装饰的呱呱叫者。

我们用静态变量跟踪所有呱呱叫次数。

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```

将Quackable当做参数传入构造器，并记录在实例变量中。

当quack()被调用时，我们就把调用委托给正在装饰的Quackable对象……

……然后把叫声的次数加一。

给装饰者加入一个静态方法，以便返回在所有Quackable中发生的叫声次数。

9 **我们需要更新此模拟器，以便创建被装饰的鸭子。**

现在，我们必须包装在QuackCounter装饰者中被实例化的每个Quackable对象。如果不这么做，鸭子就会到处乱跑而使得我们无法统计其叫声次数。

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Decorator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() + " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

每次我们创建一个Quackable，就用一个新的装饰者包装它。

公园巡逻员告诉我们，他不想计入鹅的叫声，所以我们不去装饰鹅。

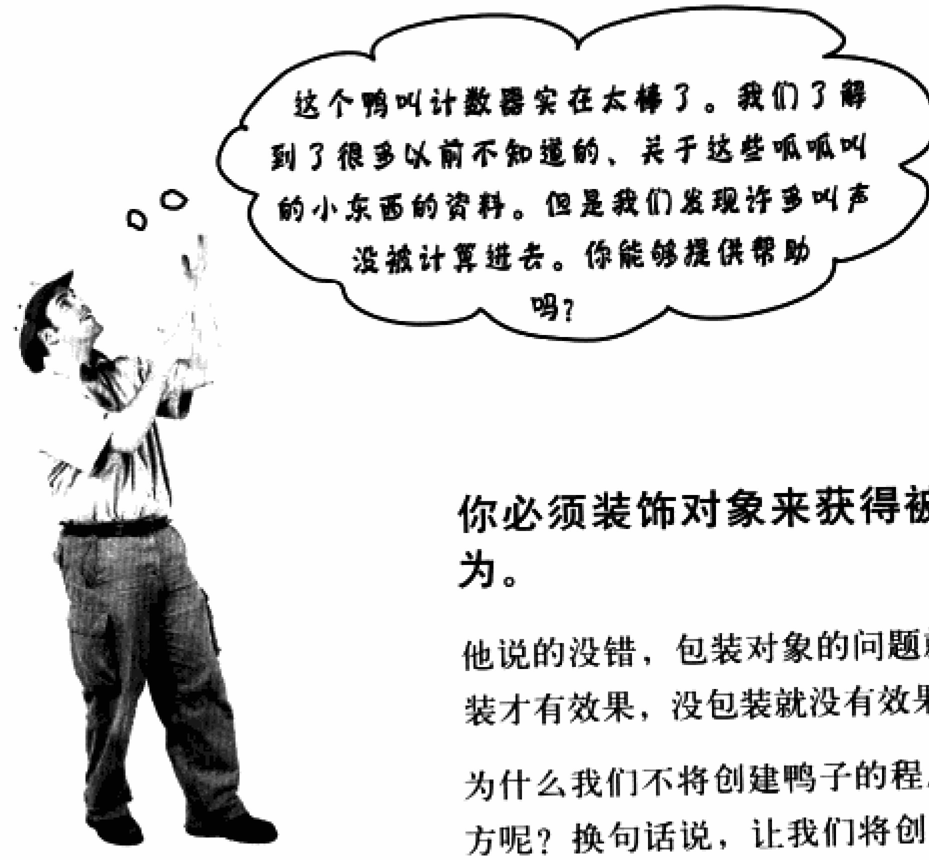
就是在这里，我们为呱呱叫学家收集呱呱叫行为。

这里没有任何的变动，被装饰的对象还是Quackable。

输出在这里！

```
File Edit Window Help DecoratedEggs
% java DuckSimulator
Duck Simulator: With Decorator
Quack
Quack
Kwak
Squeak
Honk
The ducks quacks 4 times
%
```

别忘了，鹅的叫声不计在内。



你必须装饰对象来获得被装饰过的行为。

他说的没错，包装对象的问题就是这样：有包装才有效果，没包装就没有效果。

为什么我们不将创建鸭子的程序集中在一个地方呢？换句话说，让我们将创建和装饰的部分包装起来吧。

这看起来像什么模式？

⑩ 我们需要用工厂产生鸭子！

好了！我们需要一些质量控制来确保鸭子一定是被包装起来的。我们要建造一个工厂，创建装饰过的鸭子。此工厂应该生产各种不同类型的鸭子的产品家族，所以我们要用抽象工厂模式。

让我们从AbstractDuckFactory的定义开始：

我们定义一个抽象工厂，它的子类们会创建不同的家族。

```
public abstract class AbstractDuckFactory {

    public abstract Quackable createMallardDuck();
    public abstract Quackable createRedheadDuck();
    public abstract Quackable createDuckCall();
    public abstract Quackable createRubberDuck();
}
```

每个方法创建一种鸭子。

让我们从创建一个工厂开始，此工厂创建没有装饰者的鸭子：

```
public class DuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

DuckFactory扩展抽象工厂。

每个方法创建一个产品：一种特定种类的Quackable。模拟器并不知道实际的产品是什么，只知道它实现了Quackable接口。

现在，要创建我们真正需要的工厂，CountingDuckFactory：

CountingDuckFactory也扩展自抽象工厂。

```
public class CountingDuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

每个方法都会先用叫声计数装饰者将Quackable包装起来。模拟器并不知道有何不同，只知道它实现了Quackable接口。但是巡逻员可以因此而放心，所有的叫声都会被计算进去。

⑪ 设置模拟器来使用这个工厂。

还记得抽象工厂是怎么工作的吗？我们创建一个多态的方法，此方法需要一个用来创建对象的工厂。通过传入不同的工厂，我们就会得到不同的产品家族。

我们要修改一下simulate()方法，让它利用传进来的工厂来创建鸭子。

首先，我们创建工厂，准备把它传入simulate()方法。

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

simulate()方法需要一个AbstractDuckFactory参数，利用它创建鸭子，而不是直接实例化鸭子。

这里完全没变动！

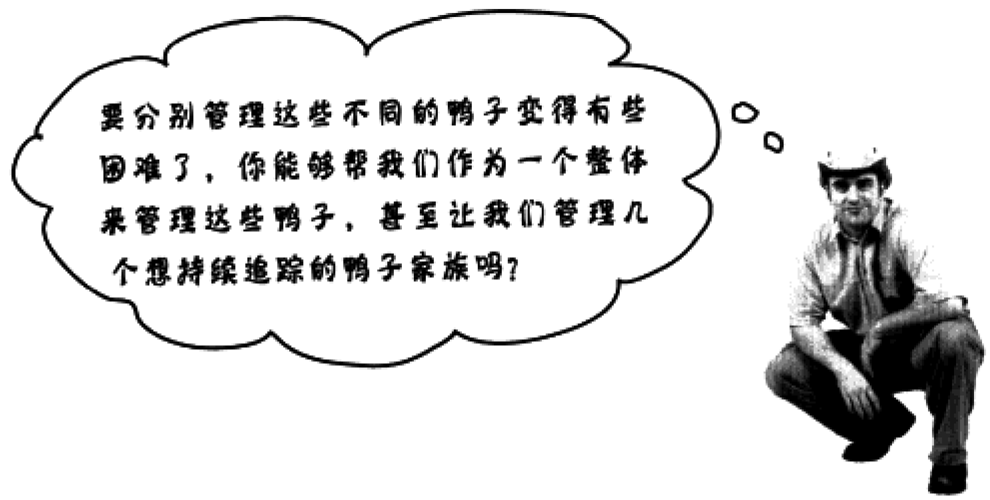
这是使用工厂的输出……

```
File Edit Window Help EggFactory
% java DuckSimulator
Duck Simulator: With Abstract Factory
Quack
Quack
Kwak
Squeak
Honk
The ducks quacks 4 times
%
```

和上一次一样，但是这次我们确定所有的鸭子都被装饰过，因为我们使用CountingDuckFactory。

Sharpen your pencil

我们仍然依赖具体类来直接实例化鹅。你能够为鹅写一个抽象工厂吗？创建“内鹅外鸭”的对象时，你怎么处理？



啊哈！他想管理一群鸭子。

巡逻员又给咱们出了个好题目：为什么我们要个别管理鸭子呢？

这还不太好管理呢！

```
Quackable mallardDuck = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
Quackable duckCall = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

simulate(mallardDuck);
simulate(redheadDuck);
simulate(duckCall);
simulate(rubberDuck);
simulate(gooseDuck);
```

我们需要将鸭子视为一个集合，甚至是子集合（subcollection），为了满足巡逻员想管理鸭子家族的要求）。如果我们下一次命令，就能让整个集合的鸭子听命行事，那就太好了。

什么模式可以帮我们？

⑫ **让我们创建一群鸭子（噢，实际上是一群Quackable）。**

还记得吗，组合模式允许我们像对待单个对象一样对待对象集合。还有什么模式能比组合模式创建一群Quackable更好呢！

让我们逐步地看这是如何工作的：

别忘了，组合需要和叶节点元素一样实现相同的接口。这里的“叶节点”就是Quackable。

```
public class Flock implements Quackable {
    ArrayList quackers = new ArrayList();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = (Quackable)iterator.next();
            quacker.quack();
        }
    }
}
```

在每一个Flock内，我们使用ArrayList记录属于这个Flock的Quackable对象。

用add()方法新增Quackable对象到Flock。

毕竟Flock也是Quackable，所以也要具备quack()方法，此方法会对整群产生作用，我们遍历ArrayList调用每一个元素上的quack()。

再靠近一点

你注意到了吗？我们其实还偷偷用了另一个设计模式，只是没有告诉你。

```
public void quack() {
    Iterator iterator = quackers.iterator();
    while (iterator.hasNext()) {
        Quackable quacker = (Quackable)iterator.next();
        quacker.quack();
    }
}
```

就是这个！迭代器模式！

13 **现在我们需要修改模拟器。**

我们的组合已经准备好了，我们需要一些让鸭子能进入组合结构的代码。

```
public class DuckSimulator {
    // 这里是主要方法

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
        System.out.println("\nDuck Simulator: With Composite - Flocks");

        Flock flockOfDucks = new Flock();

        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);

        System.out.println("\nDuck Simulator: Mallard Flock Simulation");
        simulate(flockOfMallards);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

和之前一样，创建所有的Quackable对象。

先创建一个Flock，然后把许多Quackable塞给它。这个Flock是主群。

然后创建一个新的绿头鸭群。

创建绿头鸭小家族……

……将它们加入绿头鸭群。

将绿头鸭群加入主群。

测试一整群！

只测试绿头鸭群。

最后，把数据显示给呱呱叫学家。

这里不需要修改，因为Flock也是Quackable！

执行结果……

```
File Edit Window Help FlockADuck
% java DuckSimulator
Duck Simulator: With Composite - Flocks
Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack

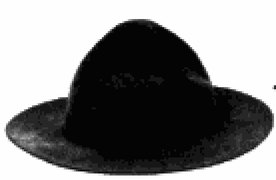
Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack

The ducks quacked 11 times
```

这里是第一群。

这是绿头鸭群。

数据看起来是对的（别忘了，鹅是不计数的）。

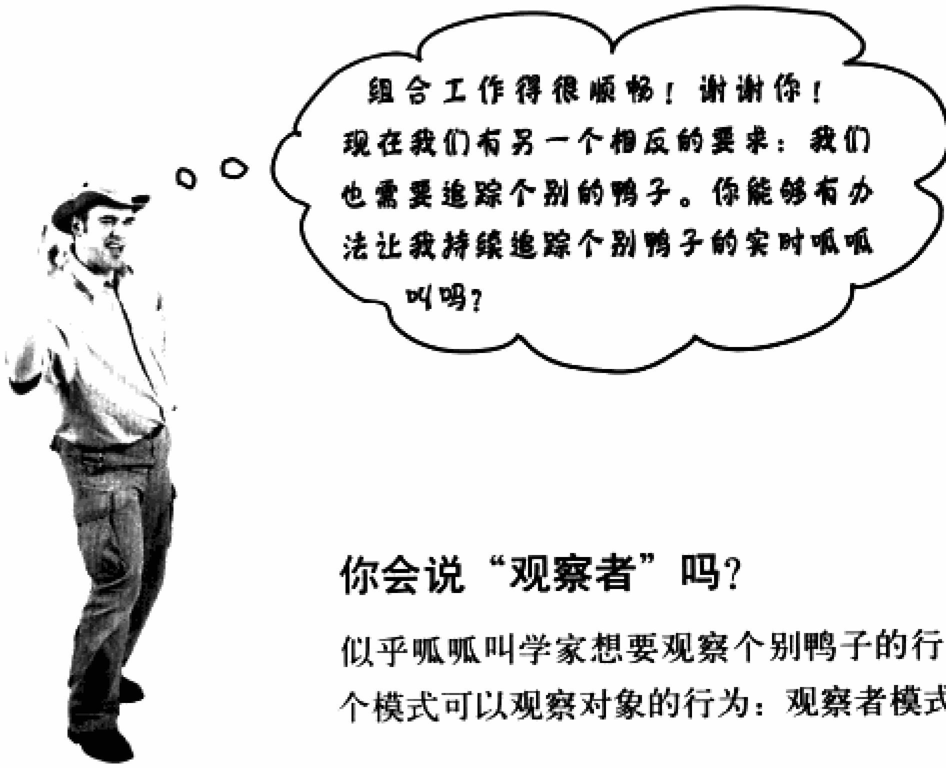


安全性 vs. 透明性

你或许还记得，在组合模式章节中，组合（菜单）和叶节点（菜单项）具有一组相同的方法，其中包括了add()方法。就因为有一组相同的方法，我们才能在菜单项上调用不起作用的方法（像通过调用add()来在菜单项内加入一些东西）。这么设计的好处是，叶节点和组合之间是“透明的”。客户根本不用管究竟是组合还是叶节点，客户只是调用两者的同一个方法。

但是在这里，我们决定把组合维护孩子的方法和叶节点分开，也就是说，我们打算只让Flock具有add()方法。我们知道给一个Duck添加某些东西是无意义的。这样的设计比较“安全”，你不会调用无意义的方法，但是透明性比较差。现在，客户如果想调用add()，得先确定该Quackable对象是Flock才行。

在OO设计的过程中，折衷一直都是免不了的，在创建你自己的组合时，你需要考虑这些。



你会说“观察者”吗？

似乎呱呱叫学家想要观察个别鸭子的行为，这让我们想起有一个模式可以观察对象的行为：观察者模式。

⑭ 首先，我们需要一个Observable接口。

所谓的Observable就是被观察的对象。Observable需要注册和通知观察者的方法。我们本来也需要删除观察者的方法，但是在这里为了让实现保持简单，我们就省略这部分了。

QuackObservable是一个接口。任何想被观察的Quackable都必须实现QuackObservable接口。

```
public interface QuackObservable {
    public void registerObserver(Observer observer);
    public void notifyObservers();
}
```

它具有注册观察者的方法，任何实现了Observer接口的对象都可以监听呱呱叫。稍后我们会定义观察者接口。

它也有通知观察者的方法。

现在我们需要确定所有的Quackable都实现此接口……

```
public interface Quackable extends QuackObservable {
    public void quack();
}
```

所以我们干脆让Quackable来扩展此接口。



QuackObservable

⑮ **现在我们必须确定所有实现Quackable的具体类都能够扮演QuackObservable的角色。**

我们需要在每一个类中实现注册和通知（同在第2章我们所做的一样）。但是这次我们要用稍微不一样的做法：我们要在另一个被称为Observable的类中封装注册和通知的代码，然后将它和QuackObservable组合在一起。这样，我们只需要一份代码即可，QuackObservable所有的调用都委托给Observable辅助类。

我们先从Observable辅助类开始下手吧……

Observable实现了所有必要的功能。我们只要把它插进一个类，就可以让该类将工作委托给Observable。

Observable必须实现QuackObservable，因为它们具有一组相同的方法。QuackObservable会将这些方法的调用转给Observable的方法。

```
public class Observable implements QuackObservable {
    ArrayList observers = new ArrayList();
    QuackObservable duck;

    public Observable(QuackObservable duck) {
        this.duck = duck;
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while (iterator.hasNext()) {
            Observer observer = (Observer)iterator.next();
            observer.update(duck);
        }
    }
}
```

在此构造器中，我们传进了QuackObservable。看看下面的notify()方法，你会发现当通知发生时，观察者把此对象传过去，好让观察者知道是哪个对象在呱呱叫。

这是注册观察者的代码。

这是通知用的代码。

接下来，让我们看看Quackable类是如何使用这个辅助类的……

⑯ 整合Observable辅助类和Quackable类

这应该不算太糟，我们只是要确定Quackable类是和Observable组合在一起的，并且它们知道怎样来委托工作。然后，它们就准备好成为Observable了。下面是MallardDuck的实现，其他的鸭子实现也类似。

```
public class MallardDuck implements Quackable {
    Observable observable;

    public MallardDuck() {
        observable = new Observable(this);
    }

    public void quack() {
        System.out.println("Quack");
        notifyObservers();
    }

    public void registerObserver(Observer observer) {
        observable.registerObserver(observer);
    }

    public void notifyObservers() {
        observable.notifyObservers();
    }
}
```

每个Quackable都有一个Observable实例变量。

在构造器中，我们创建一个Observable，并传入一个MallardDuck对象的引用。

当我们呱呱叫时，需要让观察者知道。

这是我们的两个QuackObservable方法。注意我们只是委托给辅助类进行。

Sharpen your pencil

我们还没有改变一个Quackable的实现，即QuackCounter装饰者。它也必须成为Observable。你何不试着写出它的代码呢？

几乎大功告成了！我们还需要把模式的Observer端完成。

我们已经实现了Observable所需要的一切，现在我们需要一些观察者（Observer）。我们先从Observer接口开始：

Observer接口只有一个方法，就是update()。它需要传入正在呱呱叫的对象（QuackObservable）。

```
public interface Observer {
    public void update(QuackObservable duck);
}
```

现在我们需要一个观察者：呱呱叫学家跑哪里去了？

我们需要实现Observable接口，否则就无法以QuackObservable注册。

```
public class Quackologist implements Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + duck + " just quacked.");
    }
}
```

Quackologist很简单，只有一个方法，update()。它打印出正在呱呱叫的Quackable对象。

Sharpen your pencil

万一呱呱叫学家想观察整个群，又该怎么办呢？这么做又会是什么意思呢？不妨这样来考虑：如果我们观察一个组合，就等于我们观察组合内的每个东西。所以，当你注册要观察某个群（flock），就等于注册要观察所有的孩子（抱歉，我是说所有呱呱叫者），这甚至还包括另一个群。

在进入后面的内容前，请你写下Flock观察者的代码……

18 **我们准备开始观察了。让我们更新模拟器，试试看：**

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {

        // 在这里创建鸭子工厂和鸭子

        // 在这里创建群

        System.out.println("\nDuck Simulator: With Observer");
        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);

        simulate(flockOfDucks);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

我们在这里所需要做的事就是创建一个Quackologist，把它注册成为一个群的观察者。

这次我们模拟整个群。

让我们试试看，了解这一切是如何工作的！

这是一个大场面的终曲。五个，不，有六个模式一同出现在这个令人惊讶的鸭子模拟器中。在没有更多麻烦的情况下，我们现在就为您呈现鸭子模拟器！

```
File Edit Window Help DucksAreEverywhere
% java DuckSimulator
Duck Simulator: With Observer
Quack
Quackologist: Redhead Duck just quacked.
Kwak
Quackologist: Duck Call just quacked.
Squeak
Quackologist: Rubber Duck just quacked.
Honk
Quackologist: Goose pretending to be a Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
The Ducks quacked 7 times.

%
```

在每一次呱呱叫后，不管是哪一种呱呱叫声，观察者都会收到通知。

呱呱叫学家依然会得到次数

there are no Dumb Questions

问：这就是复合模式？

答： 不，这只是一群模式携手合作。所谓的复合模式，是指一群模式被结合起来使用，以解决一般性问题。我们很快就会看到Model-View-Controller（模型-视图-控制器）复合模式。它是由数个模式结合起来而形成的新模式，一再地被用于解决许多设计问题。

问：所以，设计模式真正漂亮的地方在于，遇到问题时，我可以拿模式逐一地解决问题，直到所有的问题都被解决。我这样说对吗？

答： 错！我们在鸭子的例子中之所以这么做，主要的目的是展示许多模式可以合作。在真实的设计过程中，你不会想要这么做的。事实上，鸭子模拟器的许多部分都可以用模式解决，只是有一点“杀鸡焉用宰牛刀”的感觉。有时候，用好的OO设计原则就可以解决问题，这样其实就够了。

在下一章，我们将讨论更多这方面的问题。现在我只能告诉你，采用模式时必须要考虑到这么做是否有意义。绝对不能为了使用模式而使用模式。有了这样的观念，鸭子模拟器的设计看起来就显得做作。但是，这个例子有趣，而且在过程中还让我们体会到多个模式是如何携手解决一个问题的。

我们做了什么？

我们从一大堆Quackable开始……

有一只鹅出现了，它希望自己像一个Quackable。
所以我们利用适配器模式，将鹅适配成Quackable。现在你就可以调用鹅适配器的quack()方法来让鹅咯咯叫。

然后，呱呱叫学家决定要计算呱呱叫声的次数。
所以我们使用装饰者模式，添加了一个名为QuackCounter的装饰者。它用来追踪quack()被调用的次数，并将调用委托给它所装饰的Quackable对象。

但是呱呱叫学家担心他们忘了加上QuackCounter装饰者。
所以我们使用抽象工厂模式创建鸭子。从此以后，当他们需要鸭子时，就直接跟工厂要，工厂会给他们装饰过的鸭子。（别忘了，如果他们想取得没装饰的鸭子，用另一个鸭子工厂就可以！）

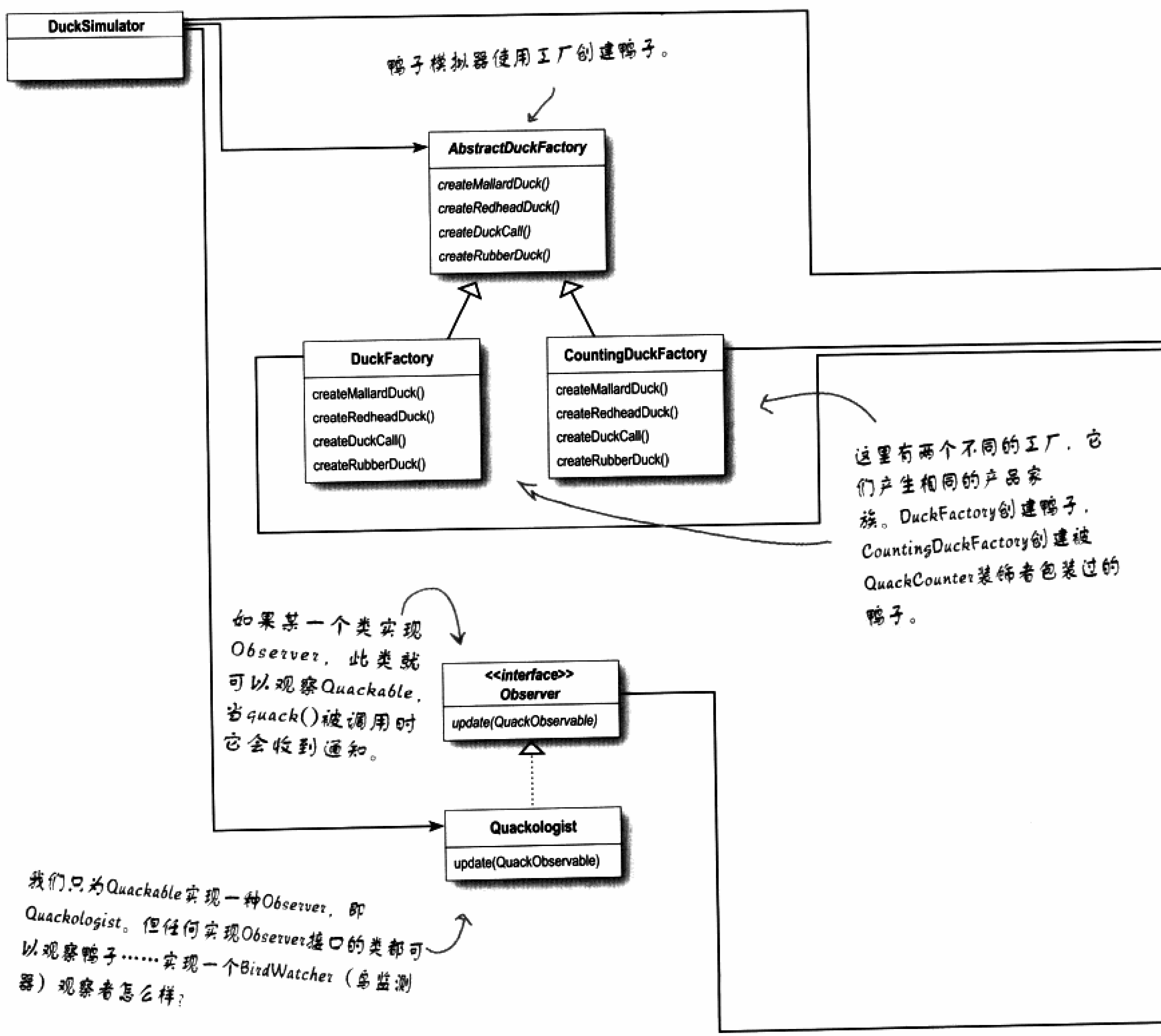
又是鸭子，又是鹅，又是quackable的……我们有管理上的困扰。
所以我们需要使用组合模式，将许多quackable集结成一个群。这个模式也允许群中有群，以便让呱呱叫家来管理鸭子家族。我们在实现中通过使用ArrayList中的java.util的迭代器而使用了迭代器模式。

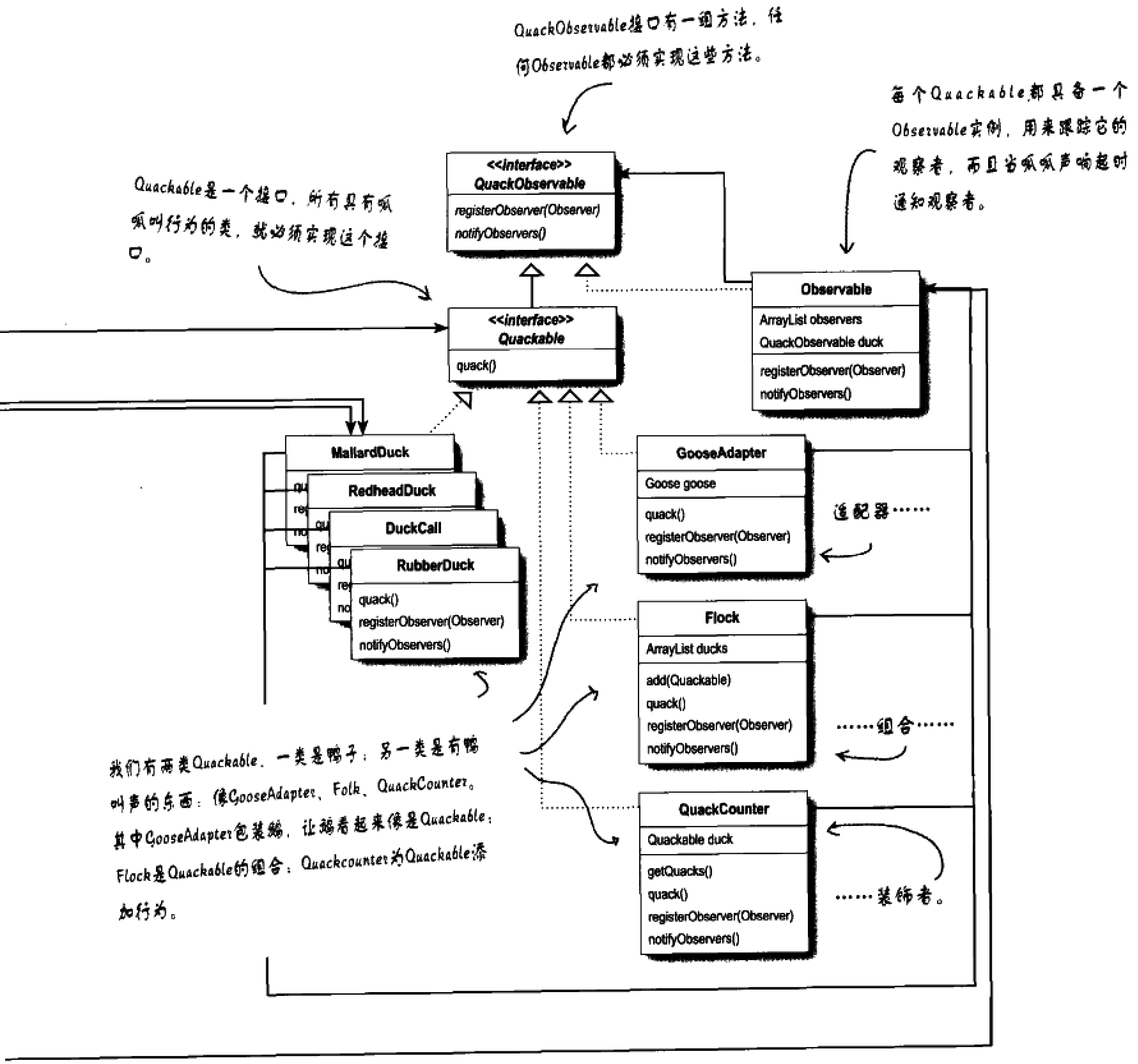
当任何呱呱声响起时，呱呱叫学家都希望能被告知。
所以我们使用观察者模式，让呱呱叫学家注册成为观察者。现在，当呱呱声响起时，呱呱叫学家就会被通知了。在这个实现中，我们再度用到了迭代器。呱呱叫学家不仅可以当某个鸭子的观察者，甚至可以当一整群的观察者。



~~鸟~~鸭瞰：类图

在一个小小的鸭子模拟器中，我们打包了许多模式。系统概览是这样的：



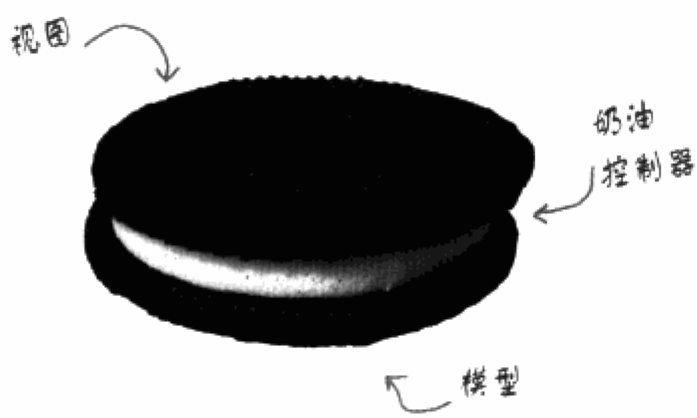

QuackObservable接口有一组方法，任何Observable都必须实现这些方法。
每个Quackable都具备一个Observable实例，用来跟踪它的观察者，而且当呱呱声响起时通知观察者。
Quackable是一个接口，所有具有呱呱叫行为的类，就必须实现这个接口。
<<interface>>
QuackObservable
registerObserver(Observer)
notifyObservers()
<<interface>>
Quackable
quack()
Observable
ArrayList observers
QuackObservable duck
registerObserver(Observer)
notifyObservers()
MallardDuck
RedheadDuck
DuckCall
RubberDuck
quack()
registerObserver(Observer)
notifyObservers()
GooseAdapter
Goose goose
quack()
registerObserver(Observer)
notifyObservers()
适配器……
Flock
ArrayList ducks
add(Quackable)
quack()
registerObserver(Observer)
notifyObservers()
……组合……
QuackCounter
Quackable duck
getQuacks()
quack()
registerObserver(Observer)
notifyObservers()
……装饰者。
我们有两类Quackable，一类是鸭子；另一类是有鸭叫声的东西：像GooseAdapter、Folk、QuackCounter。其中GooseAdapter包装鹅，让鹅看起来像是Quackable；Flock是Quackable的组合；Quackcounter为Quackable添加行为。

复合模式之王

如果猫王是复合模式，他的名字将是Model-View-Controller，他会唱这么一首歌……

模型，视图，控制器
词曲：James Dempsey

MVC是一种范型
它构造代码成为功能段，免得你脑袋淤阻
为达到复用，你必须让边界干净
这边是模型，那边是视图，控制器在中间



模型 视图，和夹心饼干一样有三层

模型 视图 控制器
模型 视图，模型 视图，模型 视图 控制器

模型对象正是你的应用系统存在的理由
你设计的对象，包含了数据、逻辑和其他
在你的应用问题域，你创建定制的类
你可以选择复用所有视图
但模型对象无需改变

你可以建模一部机器，随便什么机器。
建模一个两岁小孩
建模一瓶白葡萄酒
建模人们的窃窃私语
建模一些水煮蛋
建模Hexley的蹒跚步履

模型 视图
你可以建模GQ时尚杂志中的模特儿。
模型 视图 控制器

Java也是

视图对象通常是控件，用来显示和编辑
Cocoa①也是这样，设计良好，赢得好评
把任何老的Unicode字符串交给NSTextView②
用户可以和它交互，它几乎可以包含任何东西
但视图不知道模型
字符串可以是一个电话号码，或者亚里士多德的文学作品
保持松耦合
达到最高的复用

模型 视图，一切都是水波荡漾的蓝色③。
模型 视图 控制器

你可能正在纳闷
你可能正在纳闷
模型和视图之间的数据流动
是由控制器居中协调进行
两者之间状态的改变，数据的同步
都是由控制器控制的

---

① Cocoa是MacOS X的面向对象API。——译者
②这是Cocoa中的一个类。——译者
③这是MacOS X的用户界面Aqua预定的颜色。——译者

控制器负责将每个改变的状态送进送出

模型 视图，对于Smalltalk的人来说
这是最大的支柱
模型 视图 控制器

模型 视图，读做“噢噢”，不是“呜呜”
模型 视图 控制器

旅程尚未结束
前方还有道路
编写控制器的人
似乎没有得到掌声

模型的使命很重要
视图的外观很美妙
我或许很懒，但有时却是疯了
我写了多少代码，只是为了黏着两者
其实并不是悸痛
说穿了代码并没有神奇之处
只是用来搬动值罢了

我无意出言恐吓
但这是有声誉的
对于控制器你照做就是了

我真希望能得到一个铜板的奖赏
每次将字符串
送给TextField时

模型 视图
我们要如何丢弃黏结
模型 视图 控制器

控制器相当熟悉模型和视图
所以常常硬编码来妨碍复用
你可以将模型的键连结到任何视图的属性

一旦开始绑定
你会发现源码变少了

是的，这一切自动又免费，让我感到洋洋得意

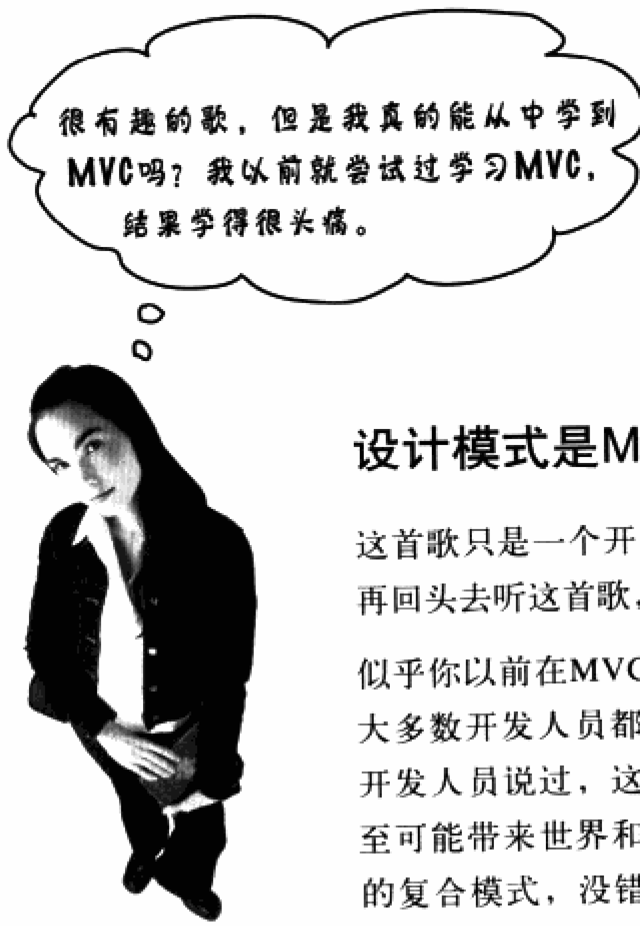
我知道一旦你使用提~~可设计工具~~ 使用 Swing
许多代码都可以自动产生
省下许多功夫

模型 视图，好处多多
模型 视图 控制器

模型 视图，但是我的应用已经交付
来不及采用MVC了
模型 视图 控制器

EAR POWER

别光是读歌词，这可是Head First系列书呀……准备好你的iPod，到下面的网址下载本歌曲：http://www.wickedlysmart.com/headfirstdesignpatterns/media.html。
靠着椅背，闭上眼睛，开始听歌吧！



设计模式是MVC的钥匙

这首歌只是一个开胃菜。你读完本章之后，再回头去听这首歌，会觉得更有趣。

似乎你以前在MVC上面遭遇过挫折？其实大多数开发人员都是这样。你可能听其他开发人员说过，这改变了他们的生活，甚至可能带来世界和平。这是一个威力强大的复合模式，没错，它虽然不能带来世界和平，但是的确可以帮助你节省编程的时间。

想要享受它的好处，就得先学会它，是吧？这次的学习经验将大大不同于以往，毕竟你现在已经懂得模式了！

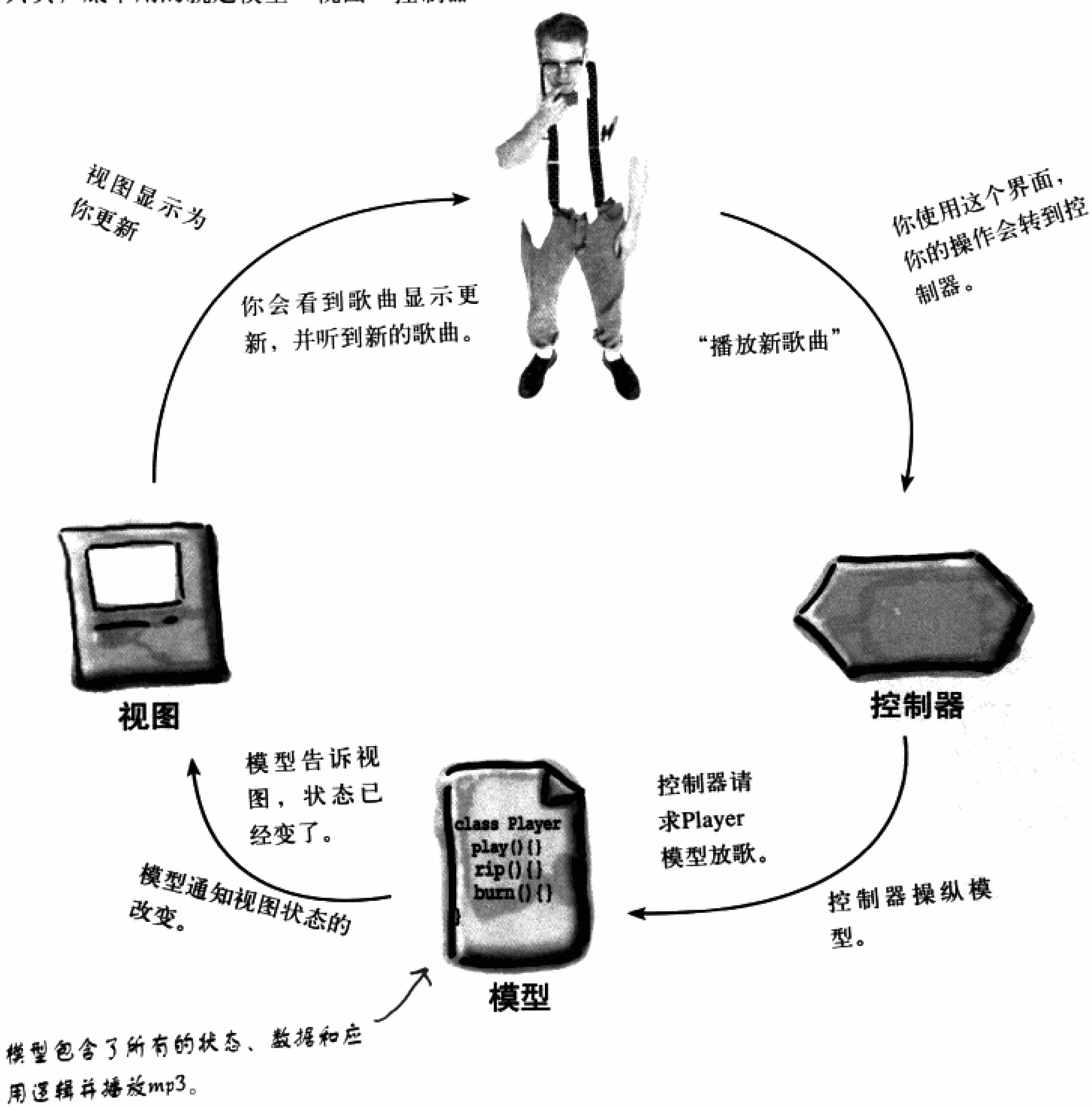
没错，设计模式是MVC的钥匙。想要由上而下地学习MVC是困难的，不是每个人都做得到。学习MVC的诀窍就在于：MVC是由数个设计模式结合起来的模式。如果你能够看着MVC内部的各个模式，MVC的一切也就会跟着明朗起来。

我们开始吧！这次，绝对不会让MVC溜掉的！

认识模型-视图-控制器

想象你正在使用你最喜欢的MP3播放器，比方说iTune。你可以用它的界面加入新的歌曲、管理播放清单、将歌曲改名。播放器有一个小型数据库，记录所有的歌曲和相关的名字和数据。播放器也可以播歌，而播歌时用户界面会显示当时的歌曲标题、运行时间等信息。

其实，底下用的就是模型－视图－控制器……



靠近一点……

MP3播放器的描述给了我们一个MVC的高层视图，但是仍然无法让我们知道复合模式内的运作细节、无法创建自己的复合模式、无法认识复合模式好在哪里。让我们从模型、视图、控制器三者的关系开始入手，然后再从设计模式的角度来看一看。

控制器

取得用户的输入并解读其对模型的意思。

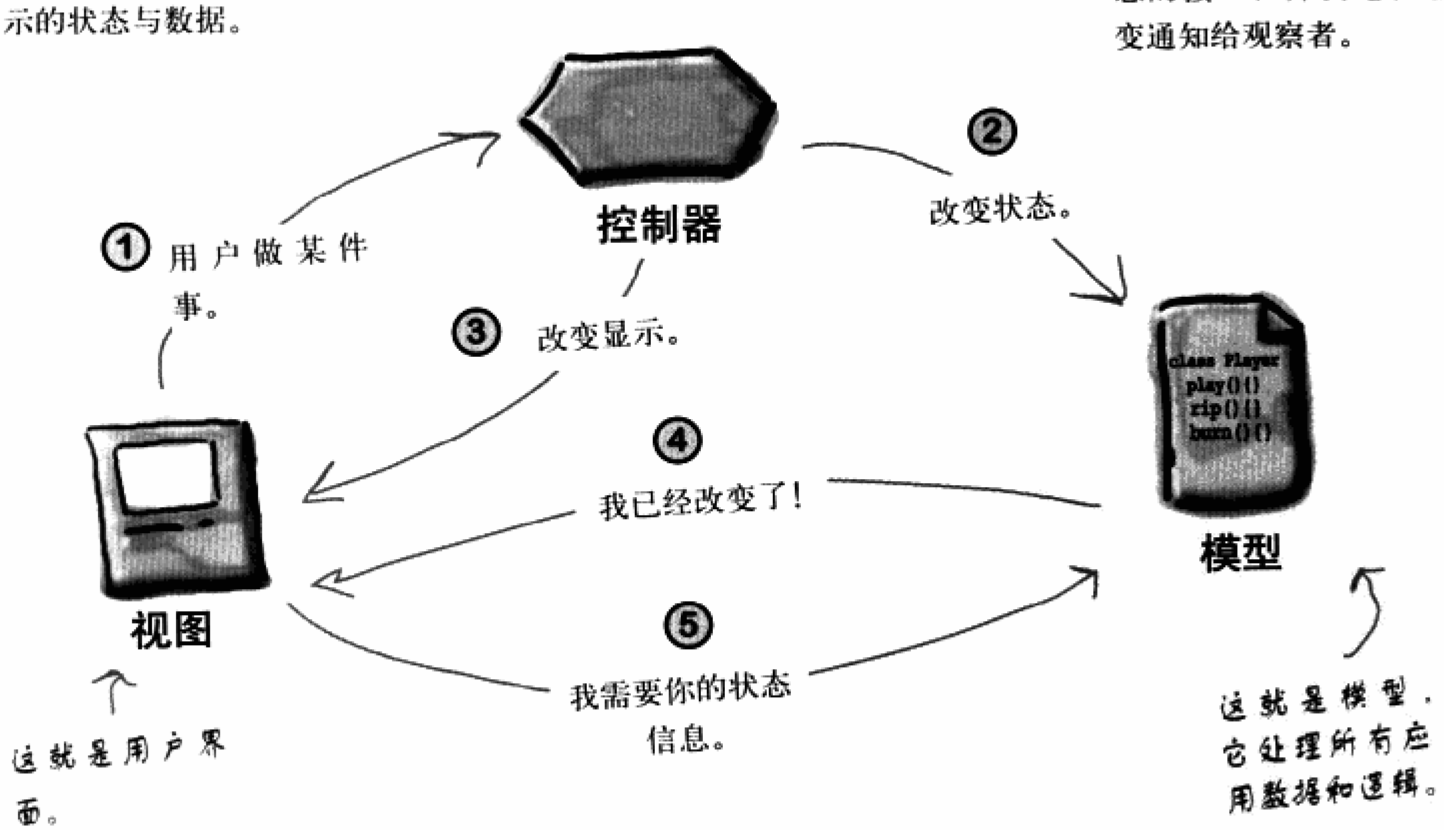
模型

模型持有所有的数据、状态和程序逻辑。模型没有注意到视图和控制器，虽然它提供了操纵和检索状态的接口，并发送状态改变通知给观察者。

视图

用来呈现模型。视图通常直接从模型中取得它需要显示的状态与数据。

这就是奶油控制器，位置在中间。



- **你是用户——你和视图交互。**

  视图是模型的窗口。当你对视图做一些事时（比方说，按下“播放”按钮），视图就告诉控制器你做了什么。控制器会负责处理。

- **控制器要求模型改变状态。**

  控制器解读你的动作。如果你按下某个按钮，控制器会理解这个动作的意义，并告知模型如何做出对应的动作。

- **控制器也可能要求视图做改变。**

  当控制器从视图接收到某一动作，结果可能是它也需要告诉视图改变其结果。比方说，控制器可以将界面上的某些按钮或菜单项变成有效或无效。

- **当模型状态改变时，模型会通知视图。**

  不管是你做了某些动作（比方说按下按钮）还是内部有了某些改变（比方说播放清单的下一首歌开始），只要当模型内的东西改变时，模型都会通知视图它的状态改变了。

- **视图向模型询问状态。**

  视图直接从模型取得它显示的状态。比方说，当模型通知视图新歌开始播放，视图向模型询问歌名并显示出来。当控制器请求视图改变时，视图也可能向模型询问某些状态。

---

there are no Dumb Questions

问：控制器可以变成模型的观察者吗？

答： 当然。在某些设计中，控制器会向模型注册，模型一有改变就通知控制器。当模型直接影响到用户界面时，就会这么做。比方说，模型内的某些状态可以支配界面的某些项目变成有效或无效，如果这样，要求视图更新相应显示其实就是控制器的事。

问：控制器所做的事情就是把用户的输入从视图发送到模型，对不对？如果只是做这些事，其实控制器没有必要存在呀！为何不把这样的代码放在视图中？大多数情况下，控制器不是只调用模型的方法吗？

答： 控制器做的事情不只有“发送给模型”，还会解读输入，并根据输入操纵模型。你真正想问的问题可能是“为何不能把这样的代码放在视图中？”你当然可以这么做，但是你不想这么做，有两个原因：首先，这会让视图的代码变得更复杂，因为这样一来视图就有两个责任，不但要管理用户界面，还要处理如何控制模型的逻辑。第二个原因，这么做将造成模型和视图之间紧耦合，如果你想复用此视图来处理其他模型，根本不可能。控制器把控制逻辑从视图中分离，让模型和视图之间解耦。通过保持控制器和视图之间松耦合，设计就更有弹性而且容易扩展，足以容纳以后的改变。

戴着模式的有色眼镜看MVC



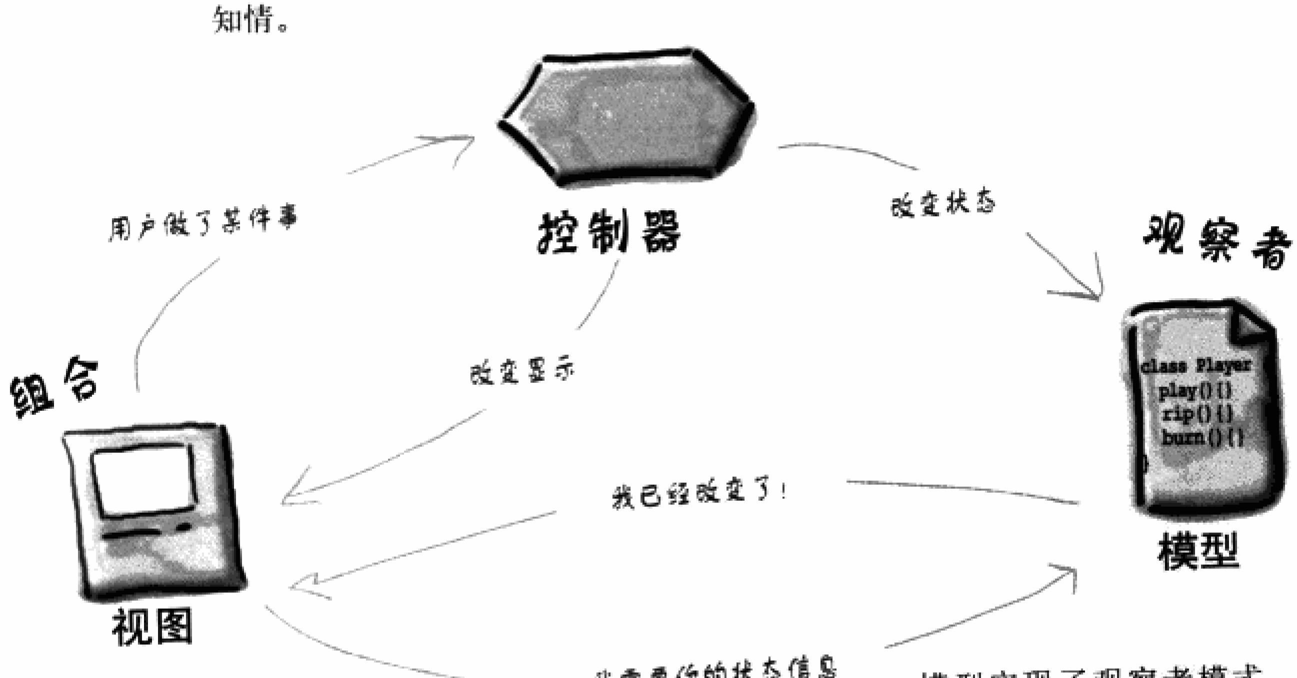
我们已经说过，学会MVC最好的方法就是看看它是由哪些模式共同组成的。

让我们先从模型开始。你可能也猜到了，模型利用“观察者”让控制器和视图可以随最新的状态改变而更新。另一方面，视图和控制器则实现了“策略模式”。控制器是视图的行为，如果你希望有不同的行为，可以直接换一个控制器。视图内部使用组合模式来管理窗口、按钮以及其他显示组件。

让我们看得更详细一点：

策略

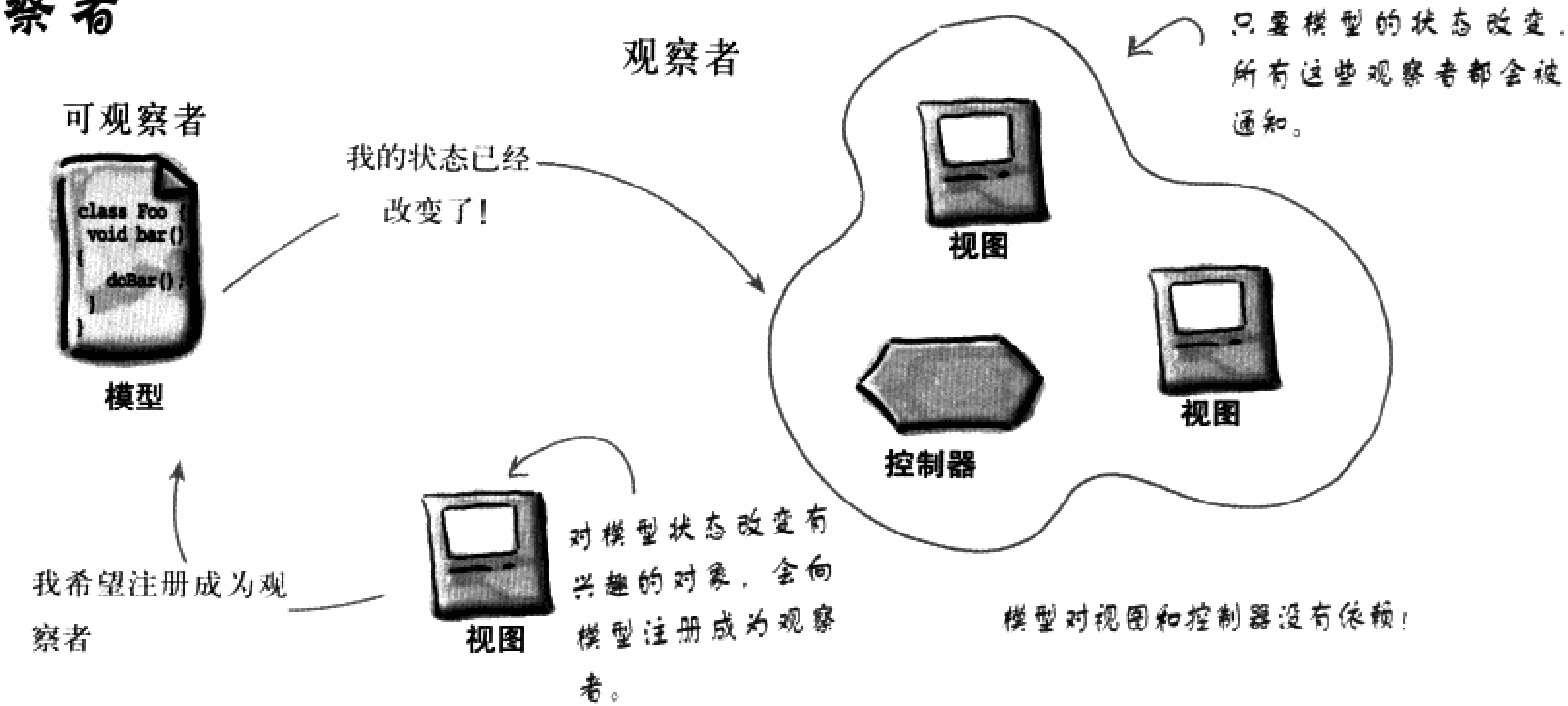
视图和控制器实现了经典的策略模式：视图是一个对象，可以被调整使用不同的策略，而控制器提供了策略。视图只关心系统中可视的部分，对于任何界面行为，都委托给控制器处理。使用策略模式也可以让视图和模型之间的关系解耦，因为控制器负责和模型交互来传递用户的请求。对于工作是怎么完成的，视图毫不知情。



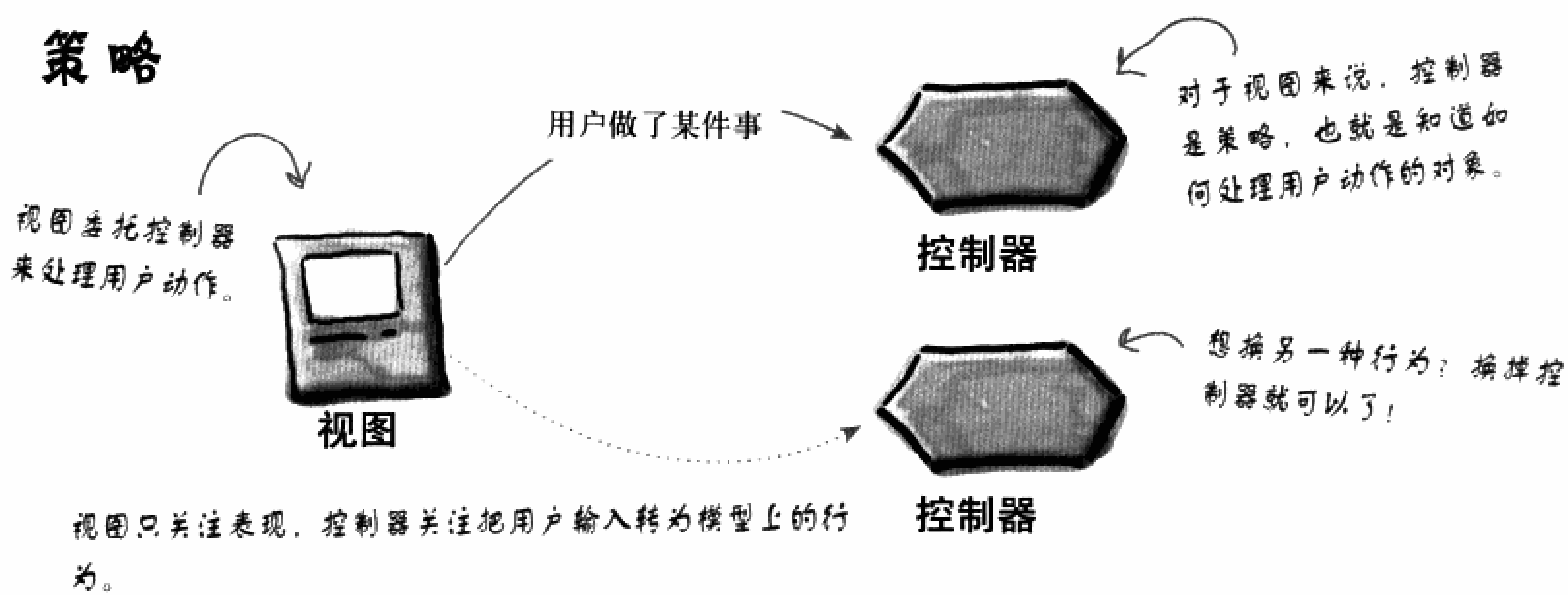
显示包括了窗口、面板、按钮、文本标签等。每个显示组件如果不是组合节点（例如窗口），就是叶节点（例如按钮）。当控制器告诉视图更新时，只需告诉视图最顶层的组件即可，组合会处理其余的事。

模型实现了观察者模式，当状态改变时，相关对象将持续更新。使用观察者模式，可以让模型完全独立于视图和控制器。同一个模型可以使用不同的视图，甚至可以同时使用多个视图。

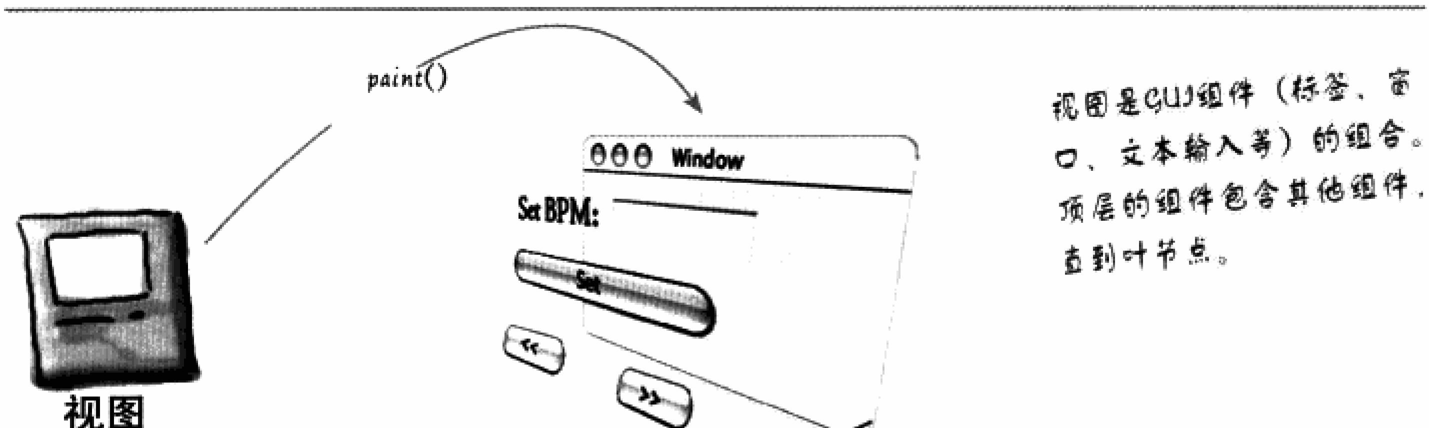
观察者


观察者
可观察者
我的状态已经改变了！
只要模型的状态改变，所有这些观察者都会被通知。
```
class Foo {
void bar()
{
doBar();
}
}
```
模型
视图
视图
控制器
我希望注册成为观察者
视图
对模型状态改变有兴趣的对象，会向模型注册成为观察者。
模型对视图和控制器没有依赖！

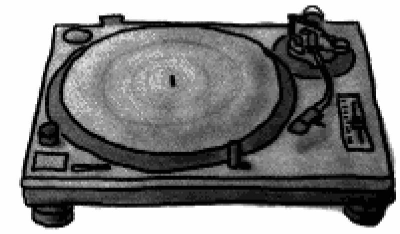
策略


用户做了某件事
对于视图来说，控制器是策略，也就是知道如何处理用户动作的对象。
视图委托控制器来处理用户动作。
控制器
视图
想换另一种行为？换掉控制器就可以了！
控制器
视图只关注表现，控制器关注把用户输入转为模型上的行为。

组合


paint()
Window
Set BPM:
Set
<<
>>
视图是GUI组件（标签、窗口、文本输入等）的组合。顶层的组件包含其他组件，直到叶节点。
视图

利用MVC控制节拍

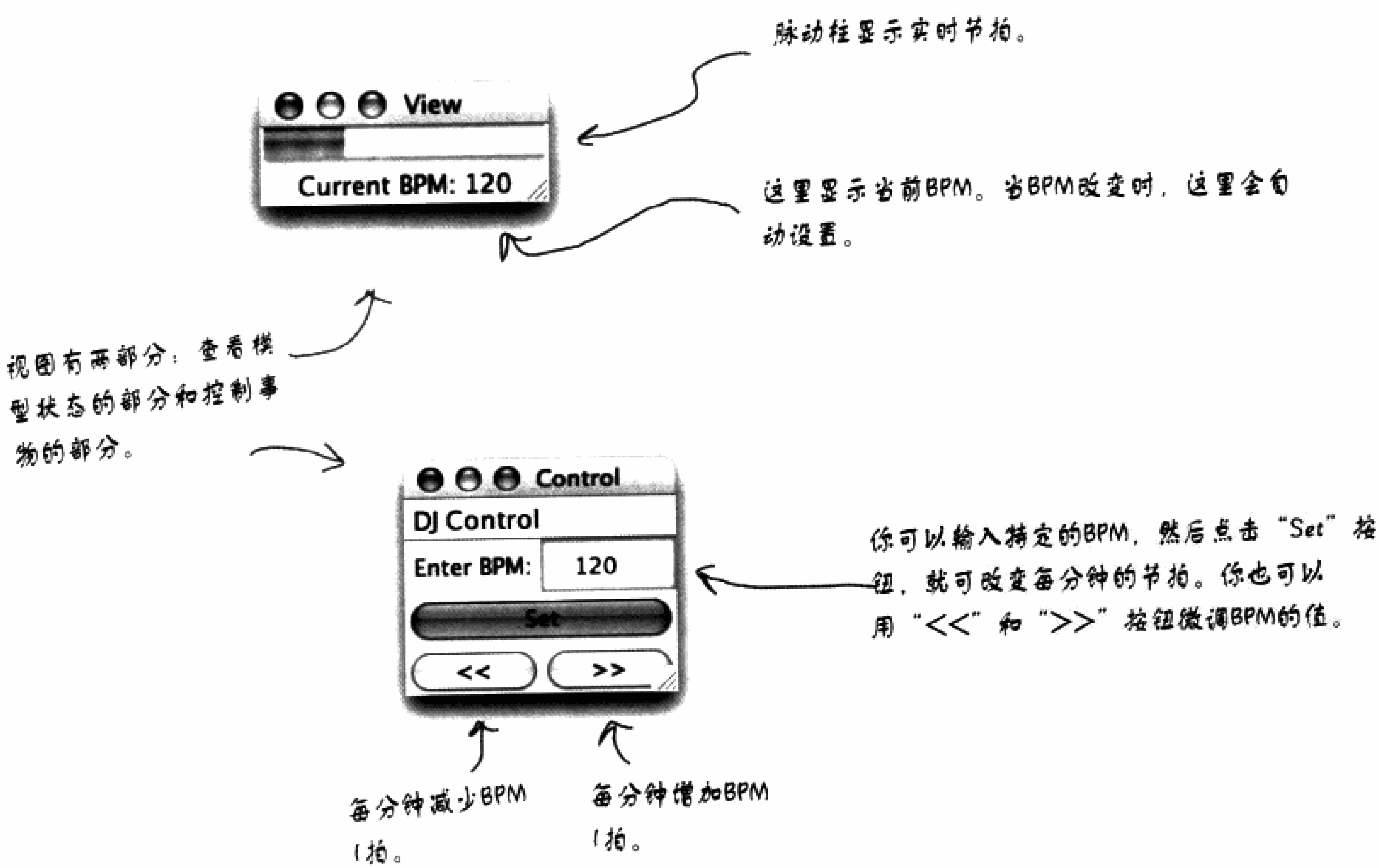


现在让你来当DJ。当DJ，节拍是头等大事，你一开始可能会用95BPM（每分钟95拍）的downtempo groove，然后转换到140BPM的trance techno，最后是80BPM的ambient mix。

这要怎么做呢？你必须控制节拍并建造工具来帮你的忙。

认识Java DJ View

让我们从这个工具的视图开始。这个视图可以让你产生鼓声节拍，并调整其BPM……



这里还有一些控制DJ View的方法……



选择"DJ Control"
-> "Start"命令，你就可以
开始产生节拍。

你可以使用
"Stop"按钮
停止产生节拍。



注意，直到你开始产生节拍，Stop都是无效的。

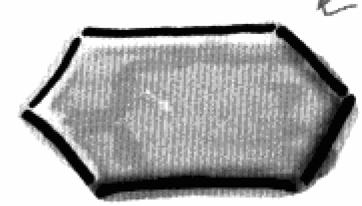
注意，节拍产生后，Start是无效的。

所有用户的动作都被送到控制器。

控制器在中间……

控制器位于视图和模型之间。它将用户的输入（比方说：从DJ控制菜单中选择"Start"），转给模型做动作，启动节拍的产生。

控制器取得输入，了解怎么一回事，然后再对模型做出请求。

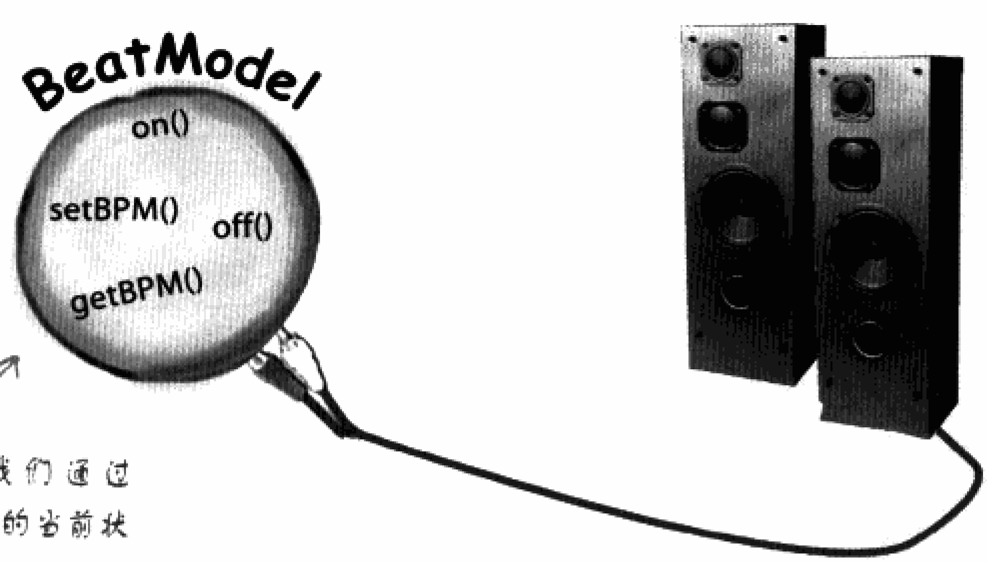


控制器

别忘了在下面的模型……

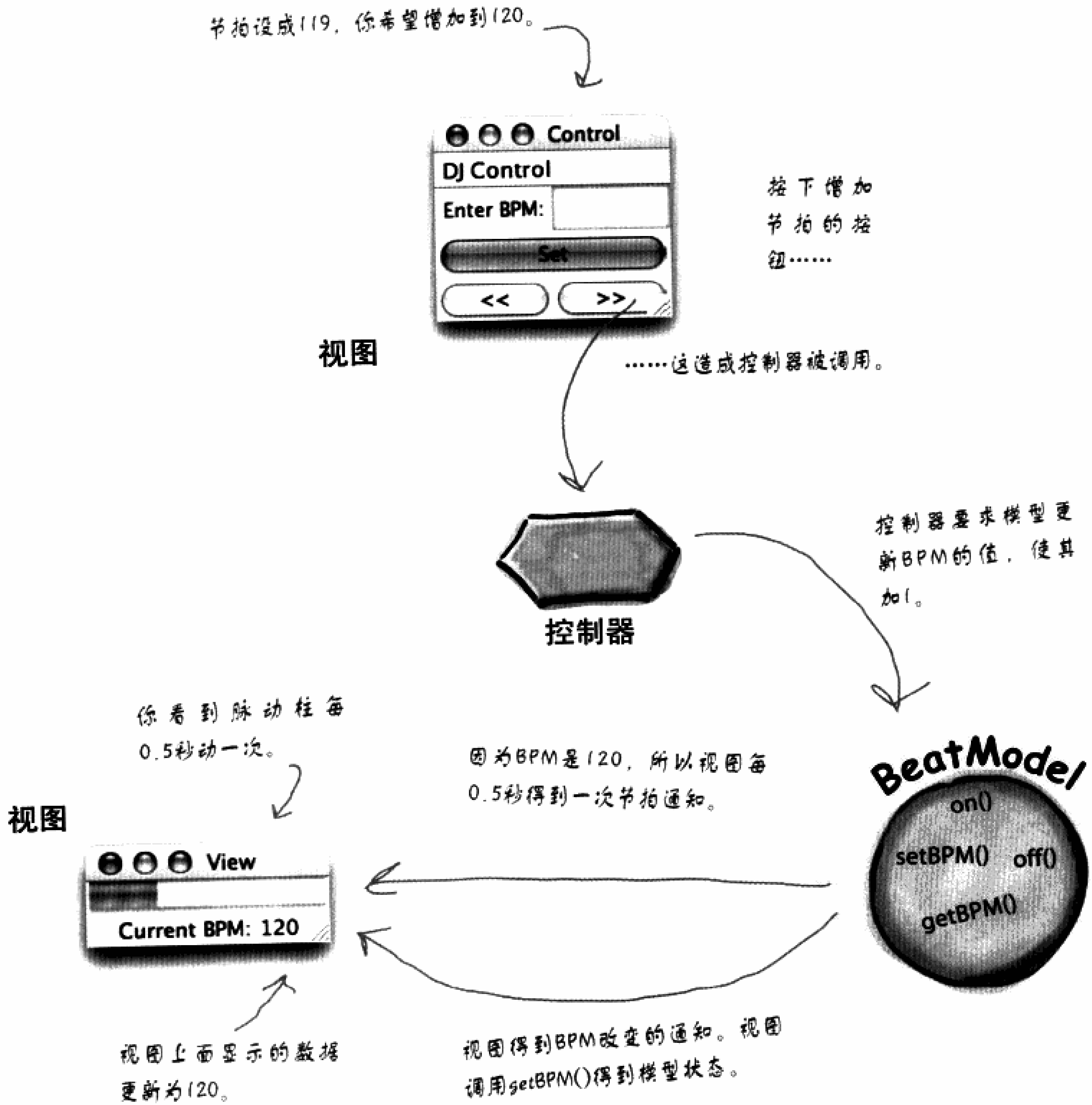
你看不到模型，但是可以听得到它。模型在背后默默地工作，管理节拍并用MIDI驱动喇叭。

BeatModel是这个系统的核心。它实现了节拍开始与停止的逻辑、管理BPM并产生声音。



该模型也可以让我们通过setBPM()方法取得它的当前状态。

把片段拼起来



节拍设成119，你希望增加到120。
Control
DJ Control
Enter BPM:
Set
<<
>>
按下增加节拍的按钮……
视图
……这造成控制器被调用。
控制器
控制器要求模型更新BPM的值，使其加1。
BeatModel
on()
setBPM()
off()
getBPM()
你看到脉动柱每0.5秒动一次。
视图
因为BPM是120，所以视图每0.5秒得到一次节拍通知。
View
Current BPM: 120
视图上面显示的数据更新为120。
视图得到BPM改变的通知。视图调用setBPM()得到模型状态。

创建碎片

现在你已经知道模型是负责维护所有的数据、状态和应用逻辑。那么BeatModel又如何呢？它的主要工作是管理节拍，所以它具有维护当前BPM的状态和许多产生MIDI事件的代码，以便产生我们听到的节拍。它也暴露一个接口，让控制器操纵节拍，让视图和控制器获得模型的状态。还有，别忘了模型使用观察者模式，所以我们也需要一些方法，让对象注册为观察者并送出通知。

在看实现之前，让我们先了解一下BeatModelInterface接口

```
public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

在BeatModel被初始化之后，就会调用此方法。

这些方法是让控制器调用的。控制器根据用户的操作而对模型做出适当的处理。

用来将节拍产生器打开或关闭。

这个方法设定BPM。调用此方法后，节拍频率马上改变。

getBPM()返回当前BPM值，如果返回值为0，表示节拍器是关闭的。

这些方法允许视图和控制器取得状态，并变成观察者。

这看起来应该很熟悉，这些方法允许对象注册成为观察者。

分成两种观察者，一种观察者希望每个节拍都被通知；另一种观察者只希望BPM改变时被通知。

现在，让我们看看具体的BeatModel类：

这是MIDI代码需要的。

我们实现了BeatModelInterface。

```
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
   //其他实例变量

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    // 注册观察者、通知观察者的代码

    // 处理节拍的MIDI代码
}
```

定序器（Sequencer）对象知道如何产生真实的节拍（你听到的拍子）。

ArrayList持有两种观察者（一种观察节拍，一种观察BPM改变）。

BPM实例变量持有节拍的频率，默认值是90BPM。

此方法为我们设置定序器和节拍音轨。

此方法开始了定序器，并将BPM设定为默认值：90。

此方法通过将BPM设置为0，停止定序器。

控制器用此方法操纵节拍，它做了三件事：

(1) 设置BPM实例变量。

(2) 要求定序器改变BPM。

(3) 通知所有的BPM观察者，BPM已经改变了。

此方法只是返回BPM实例变量，该变量指示当前BPM。

这个方法并没有在BeatModelInterface中，当新的节拍开始时，MIDI代码会调用此方法。它会通知全部的beatObserver，新的节拍开始了。



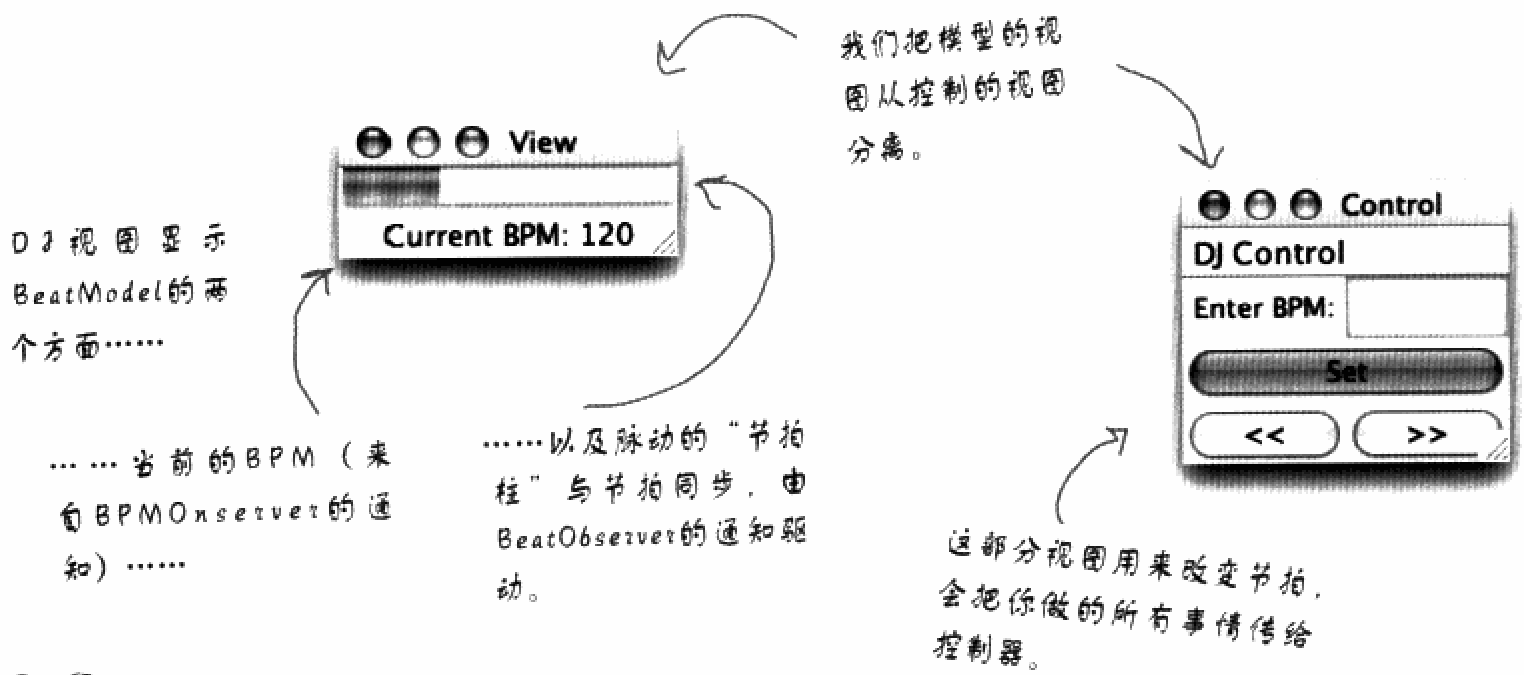
待烘烤代码

这个模型用到Java的MIDI支持来产生节拍。所有DJ类的完整实现可以从headfirstlabs.com取得，本章结尾也会列出代码。

视图

现在有趣的事情开始了，我们要把视图挂接上，使BeatModel可视化！

关于视图，我们要注意的第一件事就是实现时要用两个分离的窗口：一个窗口包含当前的BPM和脉动柱，另一个则包含界面控制。为何要这样设计？因为我们要强调包含模型视图的界面和包含其他用户控制的界面两者之间的差异。让我们详细看看视图的这两个部分：



BRAIN POWER

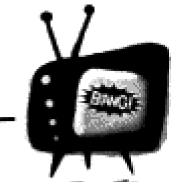
我们的BeatModel对于视图毫无所悉。这个模型是利用观察者模式实现的，当状态改变时，只要是注册为观察者的视图都会收到通知。而视图使用模型的API访问状态。我们已经实现了一种视图，你能够想出其他在BeatModel中使用通知和状态的视图吗？

基于实时节拍的灯光秀

一个基于BPM（ambient、downbeat、techno等）显示音乐风格的文本视图

实现视图

视图的两个部分（模型的视图和用户界面控制的视图）显示在两个窗口，但是属于同一个Java class。你会先看到创建模型状态的视图（显示出BPM和节拍柱）的代码，下一页会看到创建用户界面控制的代码。



注意！

这两页代码只是一个轮廓！

为了方便展示个别的功能，我们在这里将一个类分成两个，一页一个视图。但请记住，其实这两页都是属于同一个类——DJView.java。本章最后面将列出代码。

DJView是一个观察者，同时关心实时节拍和BPM的改变。

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;

    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

    public void createView() {
        // 在这里创建所有的Swing组件
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

视图持有模型和控制器的引用。控制器其实只有在控制器接口中用到，等一下你就会看到……

我们在这里创建了几个用来显示的组件。

构造器得到控制器和模型的引用，我们把它们的引用存储在实例变量中。

我们也将这个注册成为BeatObserver和BPMObserver。

模型发生状态改变时，updateBPM()方法会被调用。这时我们更新当前BPM的显示。我们可以通过直接请求模型而得到这个值。

相对地，当模型开始一个新的节拍时，updateBeat()方法会被调用。这时候，我们必须让脉动柱跳一下。我们的做法是把脉动柱设为最大值（100），让它自行处理动画部分。

继续实现视图……

现在我们来看看视图用户界面控制部分的代码。这个视图通过告诉控制器做什么来让你控制模型。别忘了，这一页的代码和上一页的代码同在一个类文件中。

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public void createControls() {
        // 在这里创建所有的Swing组件
    }
    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }
}
```



这个方法创建所有的控件，并将它们放在界面上。此方法也会处理菜单。当菜单中的Start或Stop被选中时，控制器的相应方法就会被调用。

这些方法将菜单中的Start和Stop项变成enable或disable。我们稍后会看到控制器利用这些方法改变用户界面。

点击按钮时，调用此方法。

如果Set按钮被点击，控制器就会把BPM设置成新的值。

做法类似，当点击递增或递减按钮时，该信息会传给控制器。

现在是控制器

是写丢失的片断的时候了：控制器。别忘了，控制器是策略，我们把控制器插进视图中，让视图变得聪明。

因为我们正要实现策略模式，所以从可以插进DJ View的任何策略的接口开始。我们称此接口为ControllerInterface。

视图所能够调用的控制器方法都在这里。

```
public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

在看过模型的接口后，你应该对这些方法感到熟悉。你可以开始或停止节拍，也可以改变BPM。这个接口比BeatModel的接口更"丰富"，因为你可以用"加1"或"减1"的方式调整BPM。

设计谜题

你已经看到视图和控制器一起用到了策略模式。你能把这两个类的策略模式类图绘制出来吗？

控制器的实现是这样的：

控制器实现ControllerInterface接口。

```
public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

控制器是MVC夹心饼中间的奶油，所以它必须同时和模型以及视图接触，来当两者的黏着剂。

将控制器当成参数传入创建视图的构造器中。

当用户从用户界面菜单中选择“Start”时，控制器调用模型的on()，然后改变用户界面（将start菜单项disable，将Stop菜单项enable）。

类似地，当用户从菜单中选择“Stop”时，控制器调用模型的off()，然后改变用户界面（将Start菜单项enable，将Stop菜单项disable）。

注意：控制器等于是在帮视图做决定。视图只知道如何将菜单项变成开和关，但是它并不知道在何种情况下要enable/disable。

如果被点击的按钮增加，控制器就从模型取得当前的BPM，加1，然后设置一个新的BPM。

做法和上面一样，但是当前的BPM减1。

最后，如果用户界面被用来设定任意的BPM值，控制器指示模型设置它的BPM。

全部结合在一起……



一切都准备好了，我们有模型、视图和控制器。现在就将它们整合成MVC！我们会看到、听到它们和谐地携手合作。

我们需要一点点代码才能开始，代码很短：

```
public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

先建立一个模型……

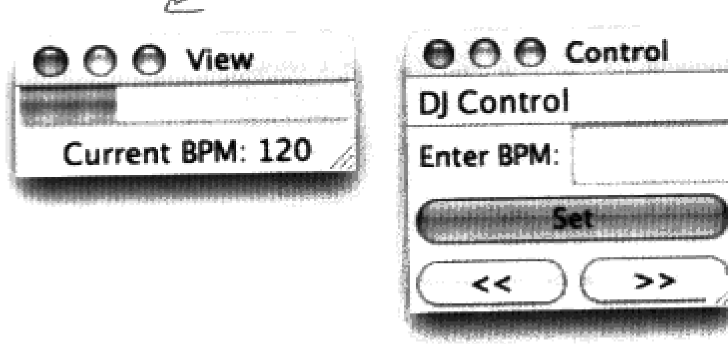
然后创建一个控制器，并将模型传给它。记住，控制器创建视图，所以我们不需要"把控制器介绍给视图认识"。

运行测试……

```
File Edit Window Help LetTheBassKick
% java DJTestDrive
%
```

运行这个……

……然后你会看到这样的画面。



要做的事

❶ 从菜单选择Start，开始产生节拍；注意控制器随后把该项disable。

❷ 使用文本输入框以及"<<"和">>"按钮来改变BPM，看看视图显示如何对改变做出反应，尽管实际上它没有逻辑链接到控件。

❸ 看看节拍柱是否一直能保持正确的拍子，因为它是模型的观察者。

❹ 播放你最喜欢的歌曲，并尝试着用"<<"或">>"按钮来增减BPM，来符合正在播放歌曲的节拍。

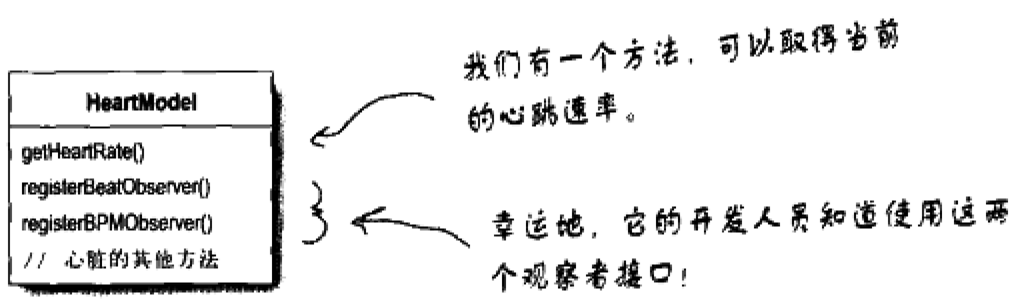
❺ 停止节拍产生器，注意控制器是如何disable。Stop菜单项和enable Start菜单项的。



探索策略

让我们更进一步地看策略模式，了解它是如何被用在MVC中的。我们也将看到另一个友好的模式常常在MVC的附近闲晃的适配器模式。

想一下DJView做了什么：它显示了节拍速率和脉动。这听起来会不会让你联想到其他事情呢？心跳？碰巧我们有一个心脏监视类，类图是这样的：



BRAIN POWER

如果能在HeartModel中复用我们当前的视图，这会省下不少功夫。但我们需要一个控制器和这个模型一同运作。还有，HeartModel的接口并不符合视图的期望，因为它的方法是getHeartRate()，而不是getBPM()。你如何设计一些类，让视图和HeartModel能够搭配使用呢？

适配模型

一开始，我们希望将HeartModel适配成BeatModel。如果不这么做，视图就无法和此模型合作，因为视图只知道getBPM()，不知道其实getHeartRate()就等于getBPM()。要怎么做？我们打算使用适配器模式，当然了！适配器其实是使用MVC时经常附带用到的技巧:使用适配器将模型适配成符合现有视图和控制器的需要的模型。

下面是将HeartModel适配成BeatModel的代码：

```
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }
    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

我们需要实现目标接口，在本例中就是BeatModelInterface。

我们在这里存储HeartModel的引用。

我们不知道这些方法将对心脏做些什么，但是看起来很可怕。所以我们让这些方法“无操作”。

当getBPM()被调用时，我们只是把它转换到HeartModel的getHeartRate()。

我们不希望对心脏做这种事，所以再次地让此方法“无操作”。

这是我们的观察者方法，直接委托给所包装的HeartModel即可。

现在我们准备写HeartController

写完了HeartAdapter，我们准备创建控制器，并让视图和HeartModel整合起来。这就是复用。

就像BeatController所做的一样，HeartController实现了ControllerInterface。

```
public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

和以前一样，控制器创建了视图，并让所有东西黏合起来。

有一个改变的地方：我们传入的是一个HeartModel，而不是BeatModel……

……HeartModel不能直接交给视图，必须先用适配器包装过才行。

最后，HeartController将菜单项disable，因为这些菜单项都是不需要的。

这些方法都没有实际的作用，毕竟我们不能像控制节拍机一样控制心跳。

就这样！现在写测试代码……

```
public class HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}
```

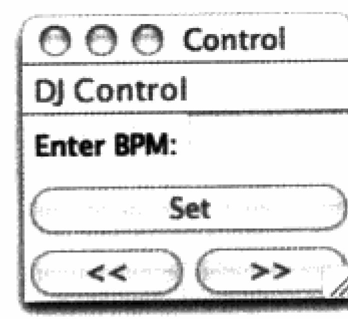
我们所需要做的就是要创建一个控制器，并传入一个HeartModel。

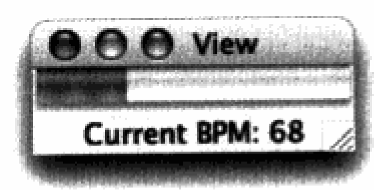
运行测试程序……

```
File Edit Window Help CheckMyPulse
% java HeartTestDrive
%
```

运行这个……

……你会看到这样的画面。





健康人的心跳速率

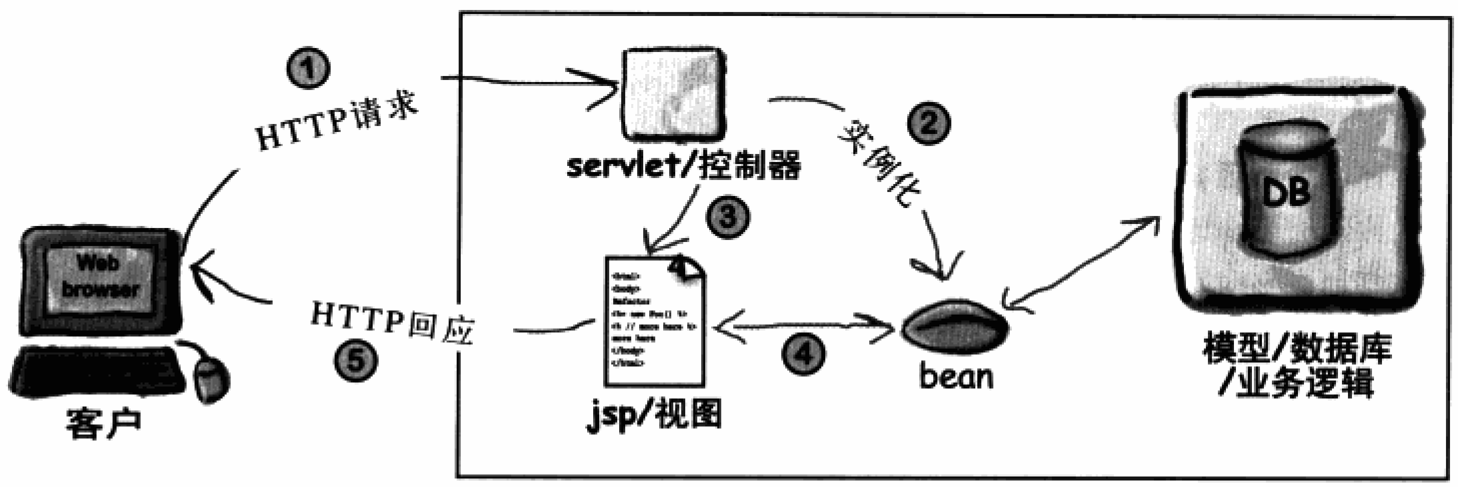
要做的事

1. 注意显示用在心跳上是没问题的！节拍柱看起来就像是心律。因为HeartModel也支持BPM观察者和Beat观察者，所以我们可以得到节拍的更新。
2. 因为心律有自然的变化，注意显示随新的BPM而更新。
3. 每次当我们取得BPM的更新时，适配器就会把getBPM()转成getHeartRate()。
4. 不能使用Start和Stop菜单项，因为控制器禁止这两个操作。
5. 其他按钮还是可以用，只是没有效果，因为控制器对这些按钮事件的实现是“无操作”。而视图可能会为了支持这些“无操作”实现而被改变。

MVC与Web

Web开发人员也都在适配MVC，使它符合浏览器/服务器模型。我们称这样的适配为“Model 2”，并使用Servlet和JSP技术的结合，来达到MVC的分离效果，就像传统的GUI。

现在就来看Model 2是怎么工作的：



① **你发出一个会被Servlet收到的HTTP请求。**

你利用网页浏览器，发出HTTP请求。这通常牵涉到送出表单数据，例如用户名和密码。Servlet收到这样的数据，并解析数据。

② **Servlet扮演控制器。**

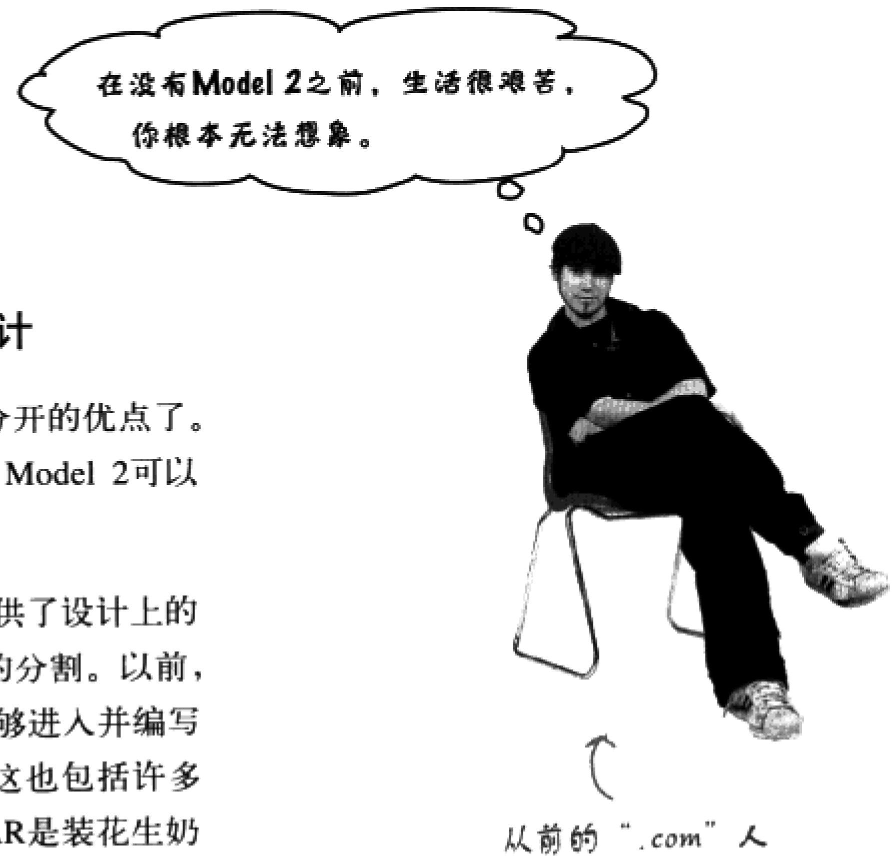
Servlet扮演控制器的角色，处理你的请求，通常会向模型（一般是数据库）发出请求。处理结果往往以JavaBean的形式打包。

③ **控制器将控制权交给视图。**

视图就是JSP，而JSP唯一的工作就是产生页面，表现模型的视图（④模型通过JavaBean中取得）以及进一步动作所需要的所有控件。

⑤ **视图通过HTTP将页面返回浏览器。**

页面返回浏览器，作为视图显示出来。用户提出进一步的请求，以同样的方式处理。



Model 2不只是一个干净的设计

你已经知道将模型、视图和控制器分开的优点了。你还需要知道“故事的其他部分”：Model 2可以帮助许多网站免于陷入混乱。

它是如何办到的呢？Model 2不仅提供了设计上的组件分割，也提供了“制作责任”的分割。以前，任何人只要能够访问你的JSP，就能够进入并编写Java代码做他们想做的事，对吧？这也包括许多不懂JAR的人（搞不好他们还以为JAR是装花生奶油酱的罐子）。我要说的重点是：许多网页制造者只懂内容和HTML，但是不懂软件。

幸好Model 2来救命了！有了Model 2，该编程的人就编程，该做网页的人就做网页，大家专业分工，责任清楚。

Model 2：你的手机也可用DJ程序



不要以为我们还没把BeatModel做成Web版，就要开溜了。其实，我们要做的是更炫的手机Web版，让你可以在手机上做DJ的工作。所以现在你可以走出DJ室，走进人群了。还等什么？让我们开始编码吧！

计划

① **修正模型。**

其实，不需要修改。现在的模型完全没问题！

② **创建Servlet控制器。**

我们需要一个简单的Servlet，可以接收HTTP请求，并对模型执行一些操作。它所需要做的是停止、开始和改变BPM。

③ **创建HTML视图。**

我们用JSP创建一个简单的视图。它会从控制器中收到一个JavaBean，从这个Bean就可以得知它所有需要显示的东西。然后JSP将产生一个HTML界面。



极客秘笈

设置你的Servlet环境



设置Servlet环境其实不在一本设计模式书的范围内，至少这本书不应该为了这个而篇幅大增。

用你的浏览器去逛一下Apache Jakarta Tomcat网页，网址在http://jakarta.apache.org/tomcat/，这里有相当详细的信息和资料。

你可能也会想要看看我们Head First系列的另一本书：Bryan Bashham、Kathy Sierra和Bert Bates所著的《Head First Servlets & JSP》。

步骤一：模型

请记得在MVC中，模型对视图和控制器一无所知。换句话说，它们之间是完全解耦的。模型只知道，有一些观察者它需要通知。这正是观察者模式美妙的地方。模型还提供一些接口，供视图和控制器获得并设置状态。

我们现在需要修改它以用于Web环境，但是由于它不依赖任何外部类，所以实在是没有什么需要修改的地方。我们可以直接使用BeatModel，真高效。直接进入步骤二吧！

步骤二：控制器Servlet

别忘了，Servlet将扮演控制器。它将收到来自Web浏览器的请求，并将其转换成作用于模型的动作。

然后，由于Web工作的方式，我们需要将一个视图返回给浏览器。所以我们需要把控制权交给视图（也就是JSP）。我们把这部分留到步骤三。

下面是Servlet的轮廓，下一页我们会看到完整的实现。

```
public class DJView extends HttpServlet {

    public void init() throws ServletException {
        BeatModel beatModel = new BeatModel();
        beatModel.initialize();
        getServletContext().setAttribute("beatModel", beatModel);
    }

        // 这里是doPost方法

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
            throws IOException, ServletException
    {
        // 实现写在这里
    }
}
```

我们扩展HttpServlet类，以便做Servlet的事情（比如接收HTTP请求）。

当Servlet第一次创建时，init()方法会被调用。

我们先创建一个BeatModel对象……

……然后传入一个beatModel的引用给ServletContext，好让ServletContext可以访问beatModel。

doGet()方法是事情真正发生的地方，下一页我们会实现此方法。

前一页的doGet()方法，是这么实现的：

我们先从ServletContext中抓取模型，稍后会用到。

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
        throws IOException, ServletException
{
    BeatModel beatModel =
        (BeatModel)getServletContext().getAttribute("beatModel");

    String bpm = request.getParameter("bpm");
    if (bpm == null) {
        bpm = beatModel.getBPM() + "";
    }

    String set = request.getParameter("set");
    if (set != null) {
        int bpmNumber = 90;
        bpmNumber = Integer.parseInt(bpm);
        beatModel.setBPM(bpmNumber);
    }

    String decrease = request.getParameter("decrease");
    if (decrease != null) {
        beatModel.setBPM(beatModel.getBPM() - 1);
    }
    String increase = request.getParameter("increase");
    if (increase != null) {
        beatModel.setBPM(beatModel.getBPM() + 1);
    }
    String on = request.getParameter("on");
    if (on != null) {
        beatModel.start();
    }
    String off = request.getParameter("off");
    if (off != null) {
        beatModel.stop();
    }

    request.setAttribute("beatModel", beatModel);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/jsp/DJView.jsp");
    dispatcher.forward(request, response);
}
```

接下来，取出所有的HTTP命令/参数……

如果命令是set，我们就找出set的值，并告诉模型。

为了递增或递减，我们从模型获得当前BPM并调整模型。

如果取得on/off命令，就告诉模型开始或停止。

控制器的责任已了，让视图接手创建HTML视图。

根据Model 2的定义，把Bean传给JSP，此Bean包含着模型的状态。但是这里的做法是：我们把真实的模型直接传给JSP，因为这个模型刚好就是一个Bean。

现在我们需要一个视图……

我们现在需要一个视图，我们的浏览器版本节拍产生器已经快完成了！在Model 2中，视图其实就是JSP。JSP只知道它会从控制器收到一个Bean。在我们的这个例子中，Bean其实就是模型，而且JSP只用到这个Bean的BPM属性。现在，JSP可以创建视图和用户界面控件了。

这就是我们的Bean，是Servlet传给我们的。

```
<jsp:useBean id="beatModel" scope="request" class="headfirst.combined.djview.BeatModel" />

<html>
<head>
<title>DJ View</title>
</head>
<body>

<h1>DJ View</h1>
Beats per minutes = <jsp:getProperty name="beatModel" property="BPM" />
<br />
<hr>
<br />

<form method="post" action="/djview/servlet/DJView">
BPM: <input type=text name="bpm"
            value="<jsp:getProperty name="beatModel"
            property="BPM" />">
 

<input type="submit" name="set" value="set"><br />
<input type="submit" name="decrease" value="<<">
<input type="submit" name="increase" value=">>"><br />
<input type="submit" name="on" value="on">
<input type="submit" name="off" value="off"><br />
</form>

</body>
</html>
```

开始写HTML！

用模型Bean提取BPM属性。

现在我们产生视图，打印出当前的BPM。

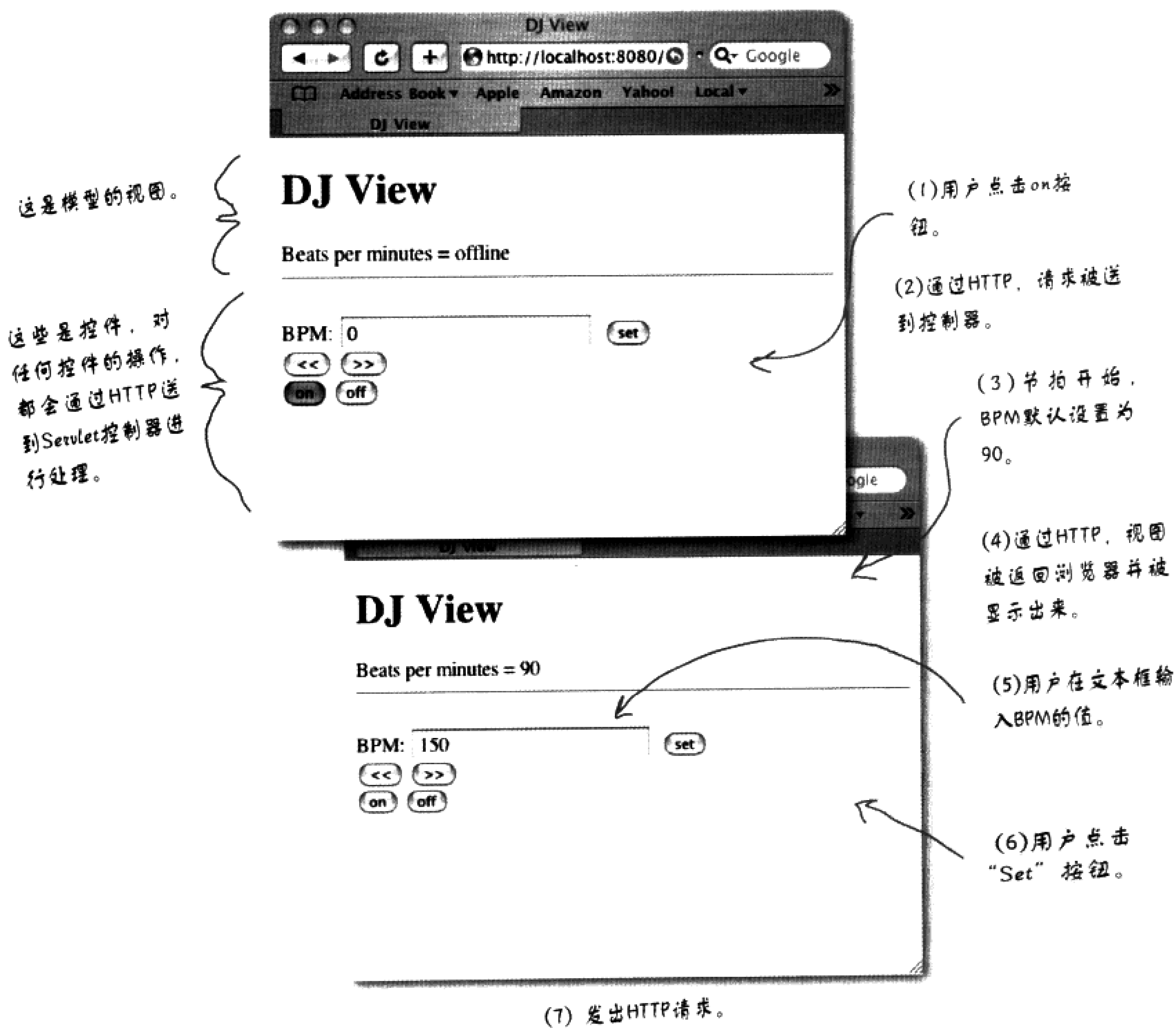
视图还具有一些控件部分。我们有一个文本输入框以及递增/递减、开/关按钮。

HTML的结束。

注意：就和MVC一样，在Model 2中，视图没有改变模型（这是控制器的工作），只使用了模型的状态。

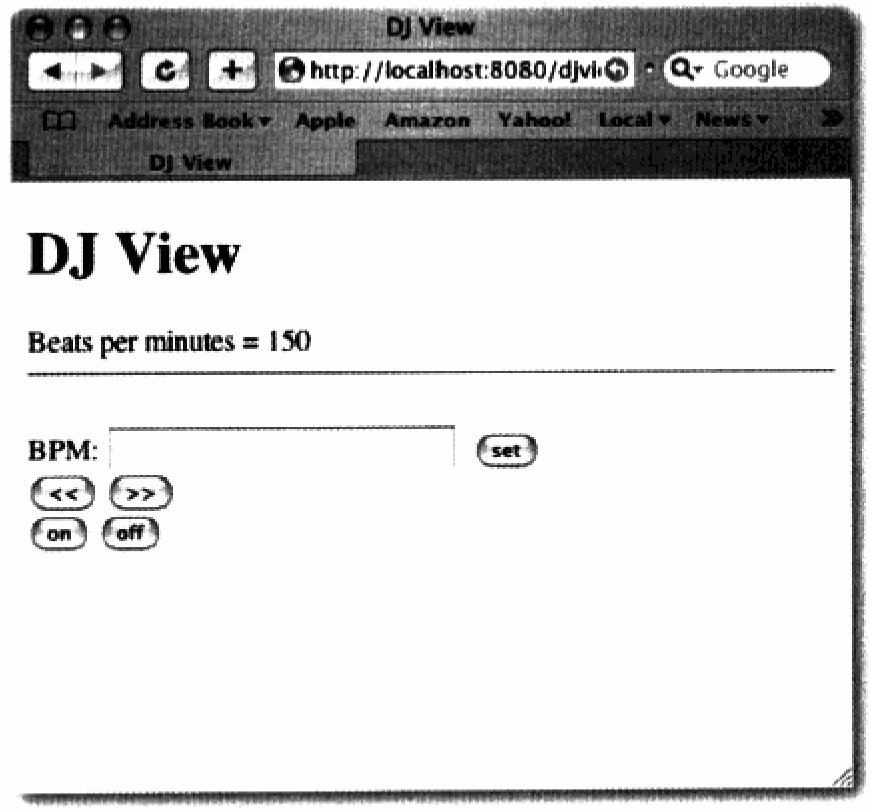
进行Model 2的测试……

打开你的Web浏览器，连到DJView Servlet……



(8)控制器把模型的BPM改成150。

(9)视图返回HTML，反映当前模型。



要做的事

1. 首先，链接到网页，你会看到BPM是0，单击“on”按钮继续。
2. 现在你会看到BPM的值是默认设置：90。你会听到Server所运行的机器上有节拍的声音。
3. 输入一个BPM值（比方说120），单击“set”按钮，网页会刷新成120BPM（你应该听到节拍加快）。
4. 利用“<<”/“>>”按钮上下调整节拍。
5. 想想看，每一步系统是如何工作的。HTML界面对Servlet（控制器）发出请求，Servlet解析用户输入，并对模型做出请求。Servlet把控制权交给JSP（视图），产生HTML视图并返回浏览器显示。

设计模式和Model 2

利用Model 2实现Web版本的DJ控制之后，你可能想知道模式去哪里了。我们的视图是JSP产生的HTML，而这个视图不再是模型的监听者。我们的控制器是Servlet，它会接收HTTP请求，但是策略模式好像不见了。至于组合模式，好像也没个影子。我们有HTML的视图显示在网页浏览器上，这还算是组合模式吗？

Model 2是MVC在Web上的调整

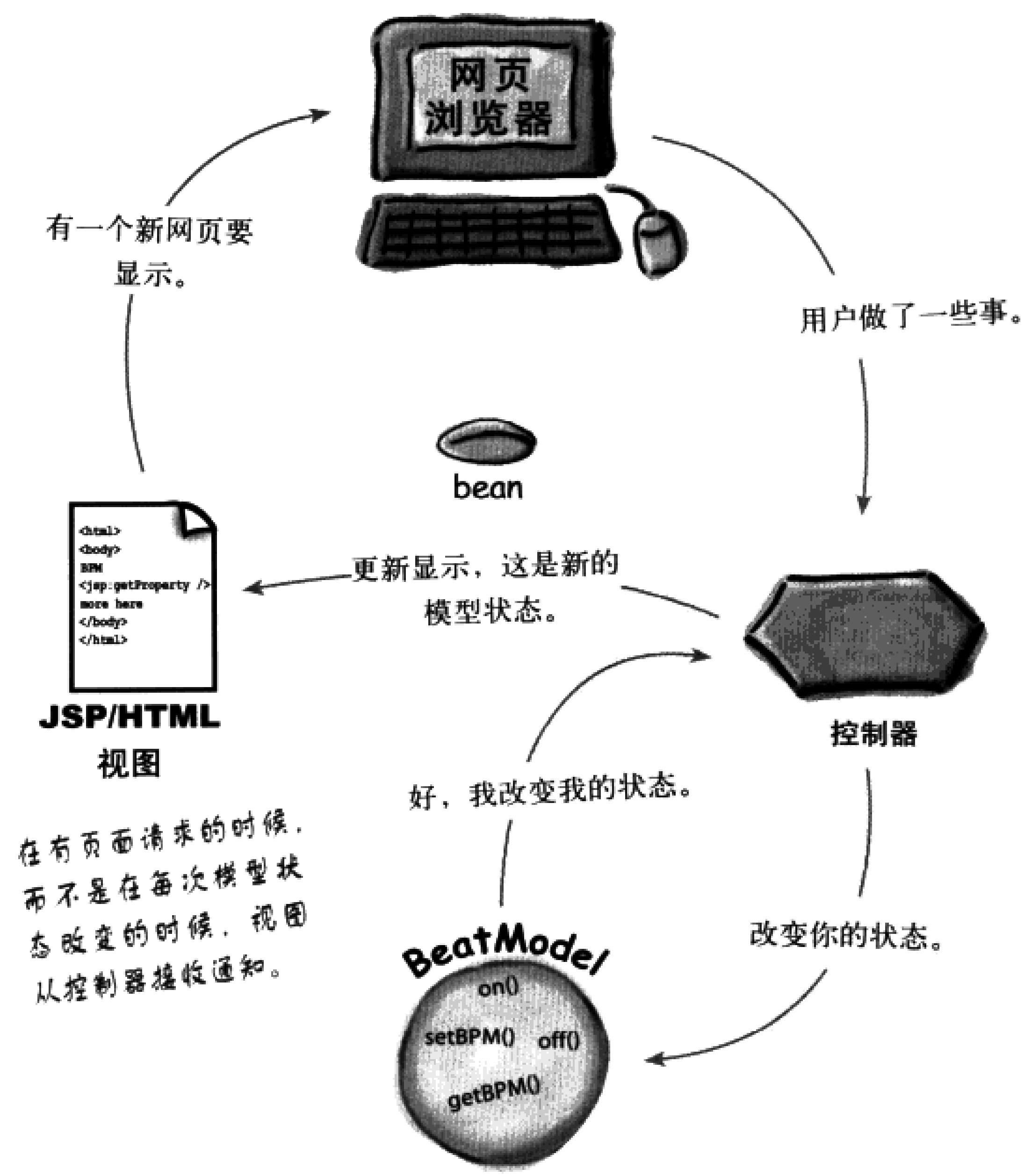
虽然Model 2看起来不像是“教科书”上的MVC，但其各部分都还在，只是为了反映Web浏览器模型的特质而经过了调整。让我们来看一看……

观察者

视图不再是经典意义上的模型的观察者，它没有向模型注册以接收状态改变通知。

但是当模型改变时，视图的确间接地从控制器收到了相当于通知的东西。控制器甚至把Bean送给视图，这允许视图可以取得模型的状态。

如果你考虑到浏览器模型，视图在HTTP响应返回到浏览器时只需要一个状态信息的更新，随时的通知是没有意义的。只有当页面被创建和返回时，创建视图并结合模型状态才有意义。

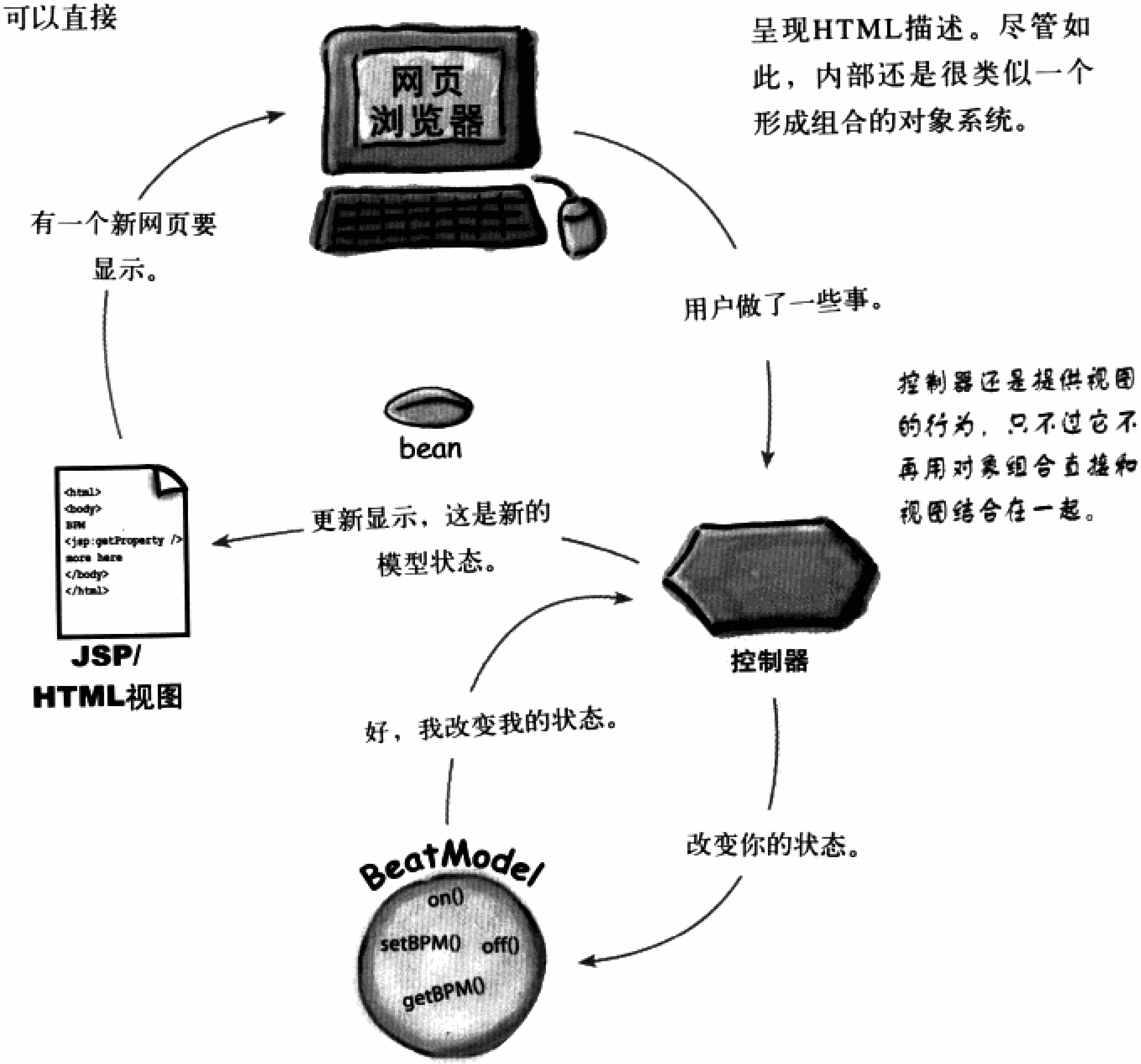


策略

在Model 2中，策略对象依然是控制器Servlet，但它不像传统的做法那样直接和视图结合。就是说，策略对象为视图实现行为，当我们想要有不同的行为时，可以直接把控制器换掉。

组合

像我们的Swing GUI，视图是利用许多图形组件一层一层叠起来的。但是在这里，则是由网页浏览器呈现HTML描述。尽管如此，内部还是很类似一个形成组合的对象系统。



there are no Dumb Questions

问：你好像有点否定组合模式在MVC中的地位。组合模式真的在MVC中吗？

答：是的，组合模式真的在MVC中。但是，这的确是一个不错的问题。今天的GUI库，像Swing，变得如此复杂，以至于我们很难注意到它的内部结构，也很难注意到它是利用组合进行构造与更新显示的。甚至，浏览器可以将标记语言转变成用户界面，这更是让我们难以想象其中牵涉到了组合。

在MVC刚刚被发现的时候，建立GUI需要许多手动干预，当时MVC模式的感受比现在更明显。

问：控制器会实现应用逻辑吗？

答：不，控制器为视图实现行为。它聪明地将来自视图的动作转成模型上的动作。模型实现应用逻辑，并决定如何响应动作。控制器也要做一些决定，决定调用哪个模型的哪个方法，但是这不能算是"应用逻辑"。应用逻辑指的是管理与操纵你的模型中的数据的代码。

问：我总是觉得"模型"这个词让我很头痛。我现在知道它是系统的重点，但是为什么要用这么模糊难懂的词汇来描述MVC的这个方面呢？

答：当取MVC名字时，他们需要一个字头为"M"的单词，否则就不能叫做MVC了。

正经一点，我们同意你的看法，一开始大家都会挠头、搞不懂模型是什么。但是大家也都逐渐地发现，除了模型，还真是找不到更恰当的词汇。

问：你说了许多关于模型的状态，这是不是意味着它用到了状态模式？

答：不，我们指的是一般意义上的状态。但的确有些模型使用状态模式管理它们的内部状态。

问：我看过有些人把MVC的控制器　描述成视图和模型之间的中介者（Mediator）。控制器有没有实现"中介者模式"？

答：我们还没有提到中介者模式（虽然你在本书的附录的模式概览中会看到），所以这里不宜说太多。大致上，中介者的意图是封装对象之间的交互，不让两个对象之间互相显式引用，以达到松耦合的目的。

因此，在某种程度上，控制器可以被视为中介者，视图不会直接设置模型的状态，而是通过控制器进行。但是，视图的确是持有用来访问模型状态的模型引用。如果控制器是彻底的中介者，那么视图就必须通过控制器才能取得模型的状态。

问：视图一定要向模型询问状态吗？为什么不在更新通知时用推送（push）模型，顺便把模型状态送过去呢？

答：当然可以在通知的时候把状态送过去，事实上，如果你再看一次JSP/HTML视图就会发现，这正是我们在做的。我们把模型状态包成Bean发送，然后视图就用Bean属性来访问状态。更早之前的BeatModel例子也可以这么做。如果你对观察者模式一章还有印象，或许还记得这么做的缺点。如果你不记得了，翻回去复习吧！

问：如果有两个以上的视图，是不是一定需要两个以上的控制器呢？

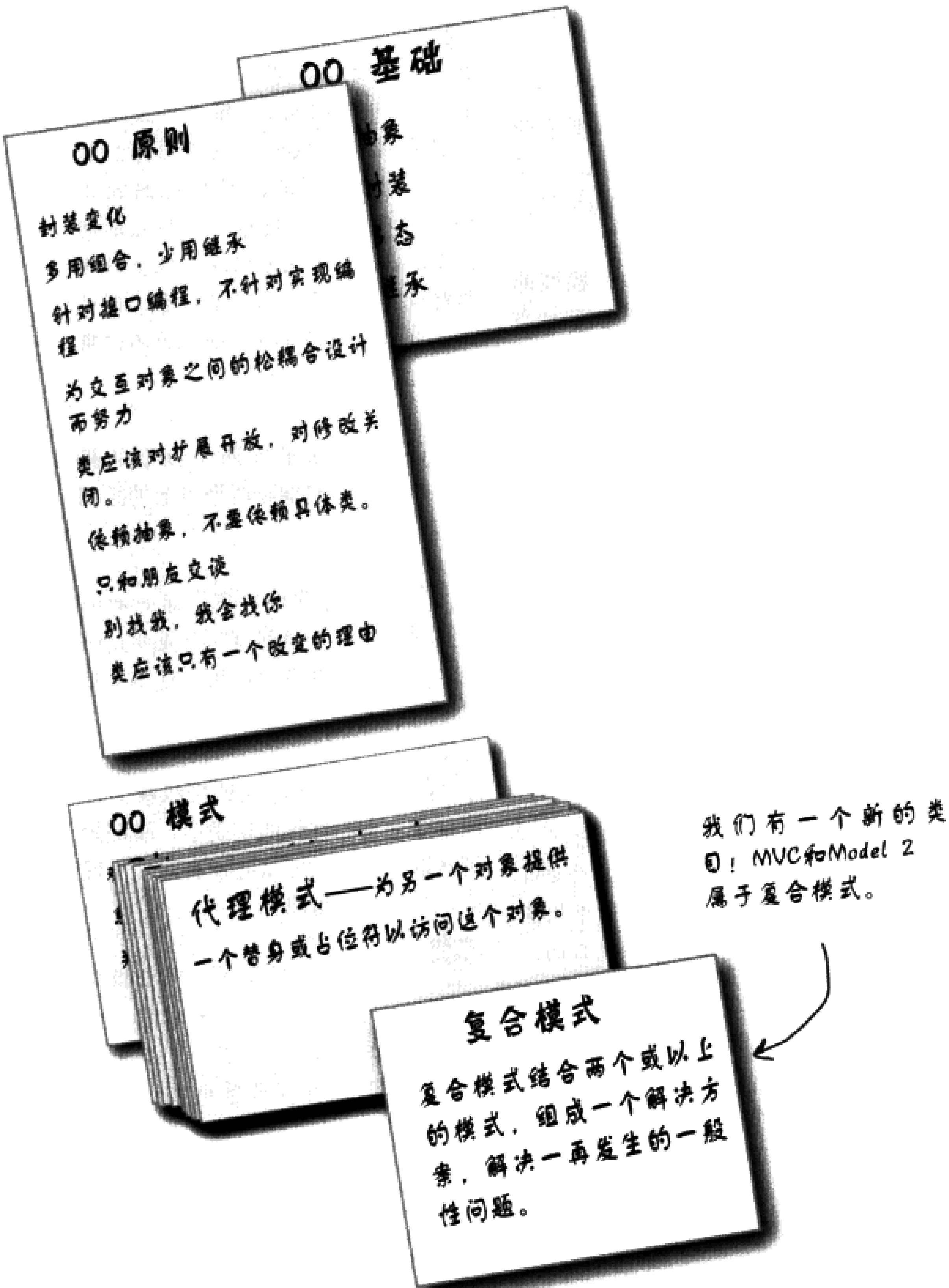
答：通常情况下，运行时一个视图搭配一个控制器；但是要让一个控制器类管理多个视图，也不是难事。

问：视图不应该操纵模型，但是我注意到在你的实现中，模型的那些改变状态的方法并没有对视图设限，这样不危险吗？

答：你说的没错，对于模型的方法，我们给视图完全的权限。这么做的原因是为了"简单"。在某些环境下，你可能只给视图访问模型的部分API。这是一个很棒的设计模式，允许你适配一个接口，只提供一个子集，你能够想起来是什么设计模式吗？

设计箱内的工具

你的设计工具箱会让所有人感到印象深刻。哇！你看这些原则和模式，现在甚至还有复合模式！



要点

- MVC是复合模式，结合了观察者模式、策略模式和组合模式。
- 模型使用观察者模式，以便观察者更新，同时保持两者之间解耦。
- 控制器是视图的策略，视图可以使用不同的控制器实现，得到不同的行为。
- 视图使用组合模式实现用户界面，用户界面通常组合了嵌套的组件，像面板、框架和按钮。
- 这些模式携手合作，把MVC模型的三层解耦，这样可以保持设计干净又有弹性。
- 适配器模式用来将新的模型适配成已有的视图和控制器。
- Model 2是MVC在Web上的应用。
- 在Model 2中，控制器实现成Servlet，而JSP/HTML实现视图。

习题解答

Sharpen your pencil

QuackCounter也是一个Quackable，当我们改变Quackable扩展QuackObservable时，我们不得不改变每个实现Quackable的类，包括QuackCounter。

QuackCounter也是一个Quackable，所以现在也是QuackObservable。

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter(Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }

    public void registerObserver(Observer observer) {
        duck.registerObserver(observer);
    }

    public void notifyObservers() {
        duck.notifyObservers();
    }
}
```

这是一个QuackCounter装饰的鸭子。需要真正处理Observable方法的就是它。

这部分代码和之前的QuackCounter版本一样。

这是两个QuackObservable方法，注意我们只要把调用委托给装饰的鸭子即可。

Sharpen your pencil

万一呱呱叫学家想观察整个群，又该怎么办呢？这么做又会是什么意思呢？不妨这样来考虑：如果我们观察一个组合，就等于我们观察组合内的每个东西。所以，当你注册要观察某个群（flock），就等于注册要观察所有的孩子，这甚至还包括另一个群。

Flock也是Quackable，所以现在它也是QuackObservable。

```
public class Flock implements Quackable {
    ArrayList ducks = new ArrayList();

    public void add(Quackable duck) {
        ducks.add(duck);
    }

    public void quack() {
        Iterator iterator = ducks.iterator();
        while (iterator.hasNext()) {
            Quackable duck = (Quackable)iterator.next();
            duck.quack();
        }
    }

    public void registerObserver(Observer observer) {
        Iterator iterator = ducks.iterator();
        while (iterator.hasNext()) {
            Quackable duck = (Quackable)iterator.next();
            duck.registerObserver(observer);
        }
    }

    public void notifyObservers() { }
}
```

在Flock内的Quackable对象都放在这里。

当你向Flock注册观察者时，其实等于是向Flock“内”的所有Quackable注册，不管是一只鸭子还是另一个群。

我们遍历Flock内的所有Quackable，把调用委托给每个Quackable。如果Quackable是另一个Flock，做同样的事。

每个Quackable都负责自己通知观察者，这样，Flock就不必操心了。当Flock将quack()委托给内部的每一个Quackable时，就是调用此方法的时机。

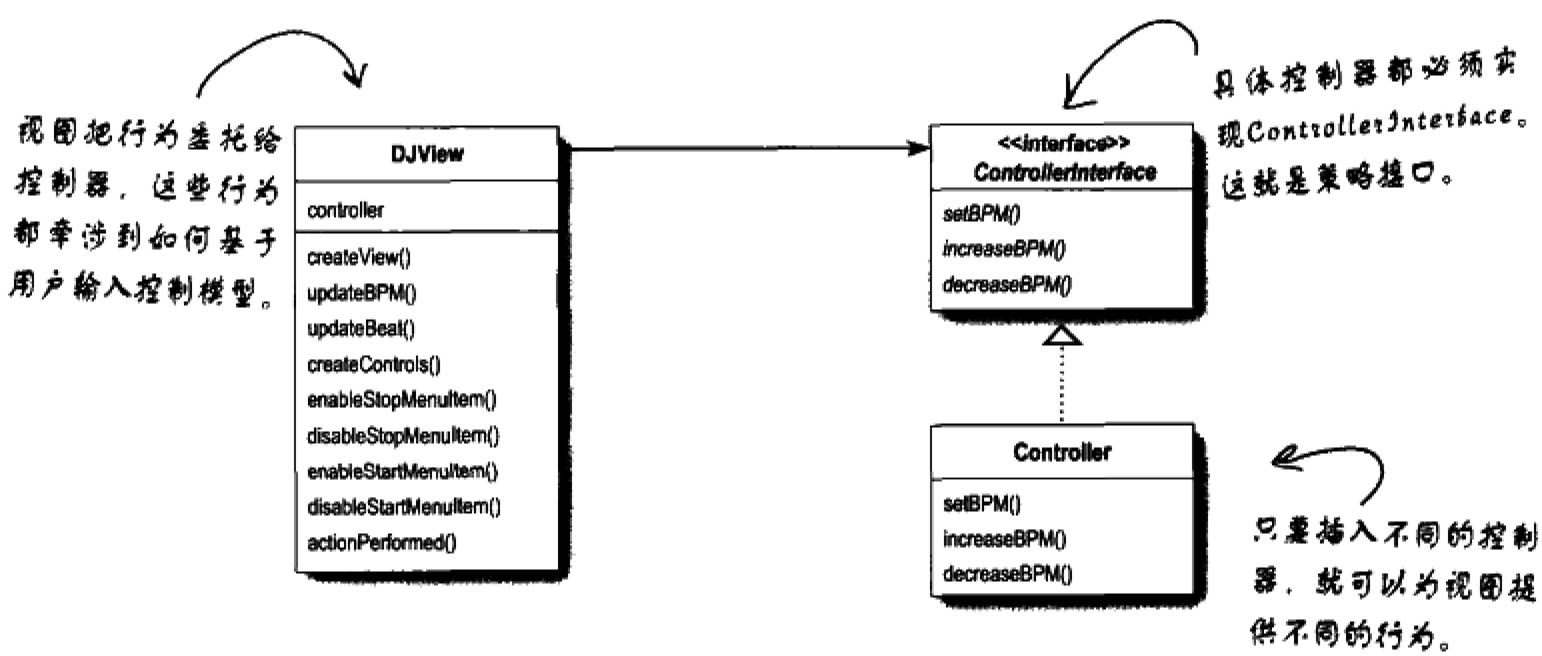
Sharpen your pencil

我们仍然依赖具体类直接实例化鹅。你能够为鹅写一个抽象工厂吗？创建“内鹅外鸭”对象时，你要怎么处理？

你可以在现有的DuckFactory类中加上createGooseDuck()方法，或者，你可以创建另一个全新的工厂，创建鹅的家族。

设计类

你已经看到视图和控制器在一起，形成策略模式，你能够把这两个类的策略模式类图绘制出来吗？





待烘烤代码

这是DJView完整的实现。包含了所有的MIDI代码来产生声音和所有的Swing组件来产生视图。你可以到http://www.wickedlysmart.com下载代码。好好玩吧！

```
package headfirst.combined.djview;

public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

节拍模型

```
package headfirst.combined.djview;

public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

```
package headfirst.combined.djview;

import javax.sound.midi.*;
import java.util.*;
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // 这里是其他的实例化变量
    Sequence sequence;
    Track track;

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }



    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }

    public void notifyBeatObservers() {
        for(int i = 0; i < beatObservers.size(); i++) {
```

待烘烤代码

```
            BeatObserver observer = (BeatObserver)beatObservers.get(i);
            observer.updateBeat();
        }
    }

    public void registerObserver(BPMObserver o) {
        bpmObservers.add(o);
    }

    public void notifyBPMObservers() {
        for(int i = 0; i < bpmObservers.size(); i++) {
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);
            observer.updateBPM();
        }
    }


    public void removeObserver(BeatObserver o) {
        int i = beatObservers.indexOf(o);
        if (i >= 0) {
            beatObservers.remove(i);
        }
    }



    public void removeObserver(BPMObserver o) {
        int i = bpmObservers.indexOf(o);
        if (i >= 0) {
            bpmObservers.remove(i);
        }
    }


    public void meta(MetaMessage message) {
        if (message.getType() == 47) {
            beatEvent();
            sequencer.start();
            setBPM(getBPM());
        }
    }

    public void setUpMidi() {
        try {
            sequencer = MidiSystem.getSequencer();
```

```
            sequencer.open();
            sequencer.addMetaEventListener(this);
            sequence = new Sequence(Sequence.PPQ,4);
            track = sequence.createTrack();
            sequencer.setTempoInBPM(getBPM());
        } catch(Exception e) {
                e.printStackTrace();
        }
    }

    public void buildTrackAndStart() {
        int[] trackList = {35, 0, 46, 0};

        sequence.deleteTrack(null);
        track = sequence.createTrack();

        makeTracks(trackList);
        track.add(makeEvent(192,9,1,0,4));
        try {
            sequencer.setSequence(sequence);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public void makeTracks(int[] list) {

       for (int i = 0; i < list.length; i++) {
          int key = list[i];

          if (key != 0) {
             track.add(makeEvent(144,9,key, 100, i));
             track.add(makeEvent(128,9,key, 100, i+1));
          }
       }
    }

    public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
            event = new MidiEvent(a, tick);

        } catch(Exception e) {
            e.printStackTrace();
        }
        return event;
    }
}
```

视图

待烘烤代码

```
package headfirst.combined.djview;

public interface BeatObserver {
    void updateBeat();
}



package headfirst.combined.djview;

public interface BPMObserver {
    void updateBPM();
}

package headfirst.combined.djview;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;
    JFrame controlFrame;
    JPanel controlPanel;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;


    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

    public void createView() {
```

```
    // 在这里创建所有的Swing组件

    viewPanel = new JPanel(new GridLayout(1, 2));
    viewFrame = new JFrame("View");
    viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    viewFrame.setSize(new Dimension(100, 80));
    bpmOutputLabel = new JLabel("offline", SwingConstants.CENTER);
    beatBar = new BeatBar();
    beatBar.setValue(0);
    JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
    bpmPanel.add(beatBar);
    bpmPanel.add(bpmOutputLabel);
    viewPanel.add(bpmPanel);
    viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER);
    viewFrame.pack();
    viewFrame.setVisible(true);
}


public void createControls() {
    // 在这里创建所有的Swing组件

    JFrame.setDefaultLookAndFeelDecorated(true);
    controlFrame = new JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.setSize(new Dimension(100, 80));

    controlPanel = new JPanel(new GridLayout(1, 2));

    menuBar = new JMenuBar();
    menu = new JMenu("DJ Control");
    startMenuItem = new JMenuItem("Start");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.start();
        }
    });
    stopMenuItem = new JMenuItem("Stop");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.stop();
            //bpmOutputLabel.setText("offline");
        }
    });
    JMenuItem exit = new JMenuItem("Quit");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    });
```



待烘烤代码

```
        menu.add(exit);
        menuBar.add(menu);
        controlFrame.setJMenuBar(menuBar);

        bpmTextField = new JTextField(2);
        bpmLabel = new JLabel("Enter BPM:", SwingConstants.RIGHT);
        setBPMButton = new JButton("Set");
        setBPMButton.setSize(new Dimension(10,40));
        increaseBPMButton = new JButton(">>");
        decreaseBPMButton = new JButton("<<");
        setBPMButton.addActionListener(this);
        increaseBPMButton.addActionListener(this);
        decreaseBPMButton.addActionListener(this);

        JPanel buttonPanel = new JPanel(new GridLayout(1, 2));

        buttonPanel.add(decreaseBPMButton);
        buttonPanel.add(increaseBPMButton);

        JPanel enterPanel = new JPanel(new GridLayout(1, 2));
        enterPanel.add(bpmLabel);
        enterPanel.add(bpmTextField);
        JPanel insideControlPanel = new JPanel(new GridLayout(3, 1));
        insideControlPanel.add(enterPanel);
        insideControlPanel.add(setBPMButton);
        insideControlPanel.add(buttonPanel);
        controlPanel.add(insideControlPanel);

        bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
        bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        controlFrame.getRootPane().setDefaultButton(setBPMButton);
        controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

        controlFrame.pack();
        controlFrame.setVisible(true);
    }

    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
```

```
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

控制器

```
package headfirst.combined.djview;

public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

待烘烤代码

```
package headfirst.combined.djview;

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

心脏模型

```
package headfirst.combined.djview;

public class HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}
```

```
package headfirst.combined.djview;
public interface HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o);
    void removeObserver(BeatObserver o);
    void registerObserver(BPMObserver o);
    void removeObserver(BPMObserver o);
}
```

```
package headfirst.combined.djview;
import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int time = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        for(;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }
            int rate = 60000/(time + change);
            if (rate < 120 && rate > 50) {
                time += change;
```

```
                notifyBeatObservers();
                if (rate != lastrate) {
                    lastrate = rate;
                    notifyBPMObservers();
                }
            }
            try {
                Thread.sleep(time);
            } catch (Exception e) {}
        }
    }
    public int getHeartRate() {
        return 60000/time;
    }

    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }

    public void removeObserver(BeatObserver o) {
        int i = beatObservers.indexOf(o);
        if (i >= 0) {
            beatObservers.remove(i);
        }
    }

    public void notifyBeatObservers() {
        for(int i = 0; i < beatObservers.size(); i++) {
            BeatObserver observer = (BeatObserver)beatObservers.get(i);
            observer.updateBeat();
        }
    }

    public void registerObserver(BPMObserver o) {
        bpmObservers.add(o);
    }

    public void removeObserver(BPMObserver o) {
        int i = bpmObservers.indexOf(o);
        if (i >= 0) {
            bpmObservers.remove(i);
        }
    }

    public void notifyBPMObservers() {
        for(int i = 0; i < bpmObservers.size(); i++) {
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);
            observer.updateBPM();
        }
    }
}
```

心脏适配器

```
package headfirst.combined.djview;
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

控制器

待烘烤代码



```
package headfirst.combined.djview;

public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

真实世界中的模式



现在你已经准备好迎接一个充满设计模式的崭新世界。但是，在你打开所有的机会大门之前，我们需要告诉你一些即将在真实世界中遇到的细节——没错，外面的世界比对象村来得复杂。来吧！从下页开始，我们会指引你的方向……



对象村指南
与设计模式相处

请接受我们的随身指南。我们在这里提供了一些技巧，让你能够和真实世界中的模式愉快相处。在这份指南中，你将会：

- ☞ 知悉所有对于“设计模式”定义的谬误。
- ☞ 发觉有哪些琳琅满目的设计模式类目，并认清为何你只需要买其中的一本。
- ☞ 避免在错误的时间点使用设计模式的尴尬。
- ☞ 学习如何将模式维持在它应属的分类中。
- ☞ 认识到发现新的模式并非只有专家做得到。阅读过我们的秘笈之后，你也有机会写下自己的模式。
- ☞ 亲眼目睹神秘的“四人组”身份被揭露。
- ☞ 能够和邻居有共同的话题——喝咖啡、聊模式。
- ☞ 学习像东方禅师一样训练你的心智。
- ☞ 通过改进你的模式词汇，结交一些朋友，并影响周围的开发人员。

定义设计模式

我敢说，在阅读完这本书之后，你已经相当了解什么是设计模式了。但我们至今还未给它一个正式的定义。你可能会对这个常用的定义感到惊讶：

> **模式**是在某情境（context）下，针对某问题的某种解决方案。

这个定义并不会让人有恍然大悟的感觉，但是别担心，我们现在就逐步了解定义中所提到的情境、问题、解决方案：

例如：你拥有一个对象的集合。

情境就是应用某个模式的情况。这应该是会不断出现的情况。

问题就是你想在某情境下达到的目标，但也可以是某情境下的约束。

你需要注意走访每个对象，而且不需理会该集合的实现。

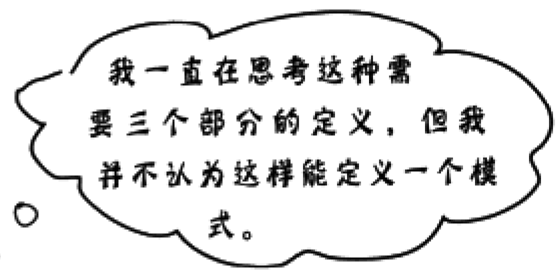
解决方案就是你所追求的：一个通用的设计，用来解决约束、达到目标。

将迭代封装进分离的类中。

这是一个需要花些时间逐步理解的定义。下面有个帮你记忆的方法：

"如果你发现自己处于某个情境下，面对着所欲达到的目标被一群约束影响着的问题，然而，你能够应用某个设计，克服这些约束并达到该目标，将你领向某个解决方案。"

现在，看起来想搞清楚什么是设计模式还需要费点功夫。毕竟，你已经知道一个设计模式是解决一个经常重复发生的设计问题。将这一切搞得如此地拘谨，究竟是为什么呢？这个嘛，一会儿你就会看到，我们采用一种规矩的方式描述模式，就能为模式创建出"类目"。而这个类目能为我们带来各种好处。





你可能是对的；让我们再多想一想……我们需要一个“问题”、一个“解决方案”和一个“情境”：

问题：我要如何准时上班？

情境：我将钥匙锁在车里了。

解决方案：打破窗户，进入车内，启动引擎，然后开车上班。

在定义中所需要的三个部分我们全都有了：我们有一个问题，这个问题包括去上班的目标，时间距离的约束，可能还有其他的影响因素；我们也具有一个情境，也就是车钥匙拿不到；我们也有一个解决方案，让我们能够取得钥匙并解决时间和空间的约束。既然这三个部分都有了，我们也就等于有了一个模式，对吧？

BRAIN POWER

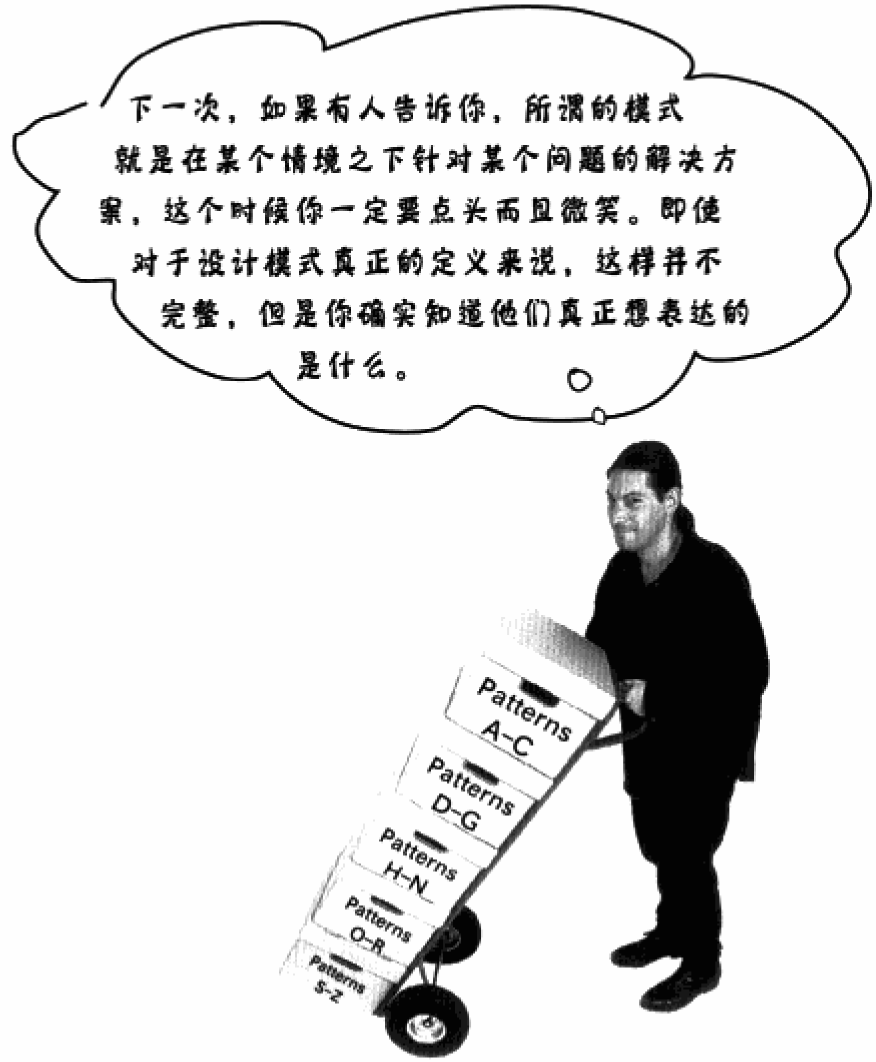
我们遵循设计模式的定义，定义了一个问题、一个情境及一个解决方案（而这个方案是行得通的！）。这是一个模式吗？如果这还不算是一个模式，那么原因是什么呢？当我们试图定义一个OO设计模式的时候，也有可能定义失败吗？

更近地观察设计模式的定义

我们的这个例子似乎符合设计模式的定义，但它不是一个真正的模式。为什么呢？我们知道模式必须应用于一个重复出现的问题。虽然一个心不在焉的人可能老是把车钥匙锁在车内，但是一再地打破车窗，这实在称不上是一个可以反复应用的解决方案（至少没有平衡另一个约束：成本）。

除了上述情况之外，它在某些方面也不符合规定。首先，别人想要在自己的特殊问题上采用这个解决方案并不容易。其次，它也违反了模式所应该具备的一个重要而简单的方面：它没有一个名字！如果没有名字，一个模式就无法变成开发人员之间共享的词汇。

幸运的是，模式并非只是被描述成简单的问题、情境和解决方案；我们有更好的方式能描述模式，并将它们收录进“模式类目”中。



问： 模式的描述是否由一个问题、一个情境及一个解决方案构成呢？

答： 通常你在模式类目中发现的模式描述不只是这些。我们很快就会看到模式类目的细节：模式类目描述某个模式的意图、动机、可能应用该模式的地方、解决方案的设计以及使用后果（好的或坏的）。

问： 稍微改变某个模式的结构以符合我的设计，这样可以吗？还是我一定要遵照严格的定义？

答： 当然你可以改变模式。像设计原则一样，模式不是法律或准则，模式只是指导方针，你可以改变模式来符合你的需要。我们也说过，真实世界中的许多实例，都不符合经典的设计模式。

然而，当你在改变模式的时候，最好能够在文档中注明它与经典的设计模式有何差异。这样一来，其他的开发人员就能够很快地认出你用的这个模式，并了解两者的差异。

问： 我要从哪里取得模式类目？

答： 第一个，也是最重要的一个设计类目是由Gamma、Helm、Johnson、Vlissides所著的《设计模式：可复用面向对象软件的基础》（Addison-Wesley 出版）。这个类目列出了23个基本的模式，再过几页我们就会谈到这本书。

还有许多其他将焦点放在不同领域（例如：企业软件、并发系统、业务系统）的模式类目。

极客秘笈

愿力与你同在

设计模式的定义告诉我们，问题包含了一个目标和一组约束。模式大师们对此有个术语，将其称为“力”。为什么？这个嘛，我们确信他们有自己的理由，但是如果你还记得那部电影：力“塑造并控制宇宙”。类似地，模式定义中的力也塑造并控制解决方案。只有当解决方案在力的两个方向中取得平衡时（光明的方向是你的目标，黑暗的方向是这些约束），这才算是有用的模式。当你第一次在模式的讨论中看到这个叫做“力”的术语时，可能感到很困惑，但只要记住，力有两个方向（目标和约束），而且需要力平衡才能够创建一个模式的解决方案。别让这个术语挡住你的路，愿力与你同在！



Frank：Jim，也算我们一份吧，我们东拼西凑地看了些文章，才学了一招半式的模式。

Jim：没问题，每个模式的类目都包含了一组模式，也描述了模式之间关系上的细节。

Joe：你是说模式的类目不只一份？

Jim：当然。有些类目是基础的设计模式，有些则是领域特定模式，例如EJB模式。

Frank：你正在看的是哪一份类目？

Jim：这是经典的四人组类目；包含了23个基础的设计模式。

Frank：四人组？

Jim：没错，四人组是四个作者的简称，他们合作写了第一本设计模式的类目。

Joe：这个类目有些什么？

Jim：有一组相关联的模式。每个模式的描述方式都遵照一个模板，并阐述该模式的许多细节。比方说，每个模式都有一个“名称”。

Frank：哇塞！模式还有名称，真不得了！

Jim：别小看名称，Frank。事实上，名称可是非常重要的呢！当每个模式都有一个名称的时候，我们谈论起模式来就相当容易了；也就是说，大家会有一个共享的词汇。

Frank：好啦！好啦！我只是在开玩笑。继续说吧，还有些什么？

Jim：就像我所说的，每个模式都要遵照一个模板。每一个模式都有名称和几节完整的叙述。例如，有一节叫做意图（Intent），描述该模式是什么，有点儿像是定义。然后还有叫做动机（Motivation）和适用性（Applicability）的节，描述何时何地该使用这个模式。

Joe：那么关于设计呢？

Jim：有几节是描述类图内的所有组成模式的类的设计，以及每个类扮演的角色。也有一节描述如何实现这个模式，而且通常有展示怎么做的范例代码。

Frank：听起来好像面面俱到。

Jim：还不只这些。还有一些例子告诉我们在真实的系统中，这个模式会使用在何处。除此之外，我认为最有用的小节之一是：此模式和其他的模式之间有何关联。

Frank：噢！你的意思是说它们会告诉你像“状态和策略有何差异”这样的东西？

Jim：没错！

Joe：那么Jim，你到底要如何使用这个类目呢？当遇到问题时，你会翻阅它来寻找解决方案吗？

Jim：首先，我试着让自己熟悉所有的模式以及它们之间的关系。然后，当我需要一个模式的时候，大概就知道是什么模式。我会参考描述动机和适用性的小节，确认我的想法没错。还有一个很重要的小节：结果。我浏览这个模式的“结果”，确保该模式不会给我的设计带来意外的影响。

Frank：听起来很有道理。一旦你知道这个模式是正确的，究竟要如何应用到你的设计中，并实现它？

Jim：这就是为什么需要类图。我先是阅读“结构”这一节，以了解类图，然后看“参与者”这一节，确定我了解每一个类的角色。接下来，就可以开始进行自己的设计，做出符合我的需求的一些更改，并继续阅读“实现/范例代码”小节，以确认我知道可能会遇到的所有较好的实现技巧。

Joe：现在我终于了解类目如何真正地帮我加快使用模式的脚步。

Frank：是的。Jim，你能带我们浏览一遍模式的描述吗？

类目中所有的模式都是以一个“名称”开始的，名称是模式中很重要的一部分——如果没有好名称，该模式就无法成为你和其他开发人员之间共享词汇的一部分。

“动机”给出了问题以及如何解决这个问题的具体场景。

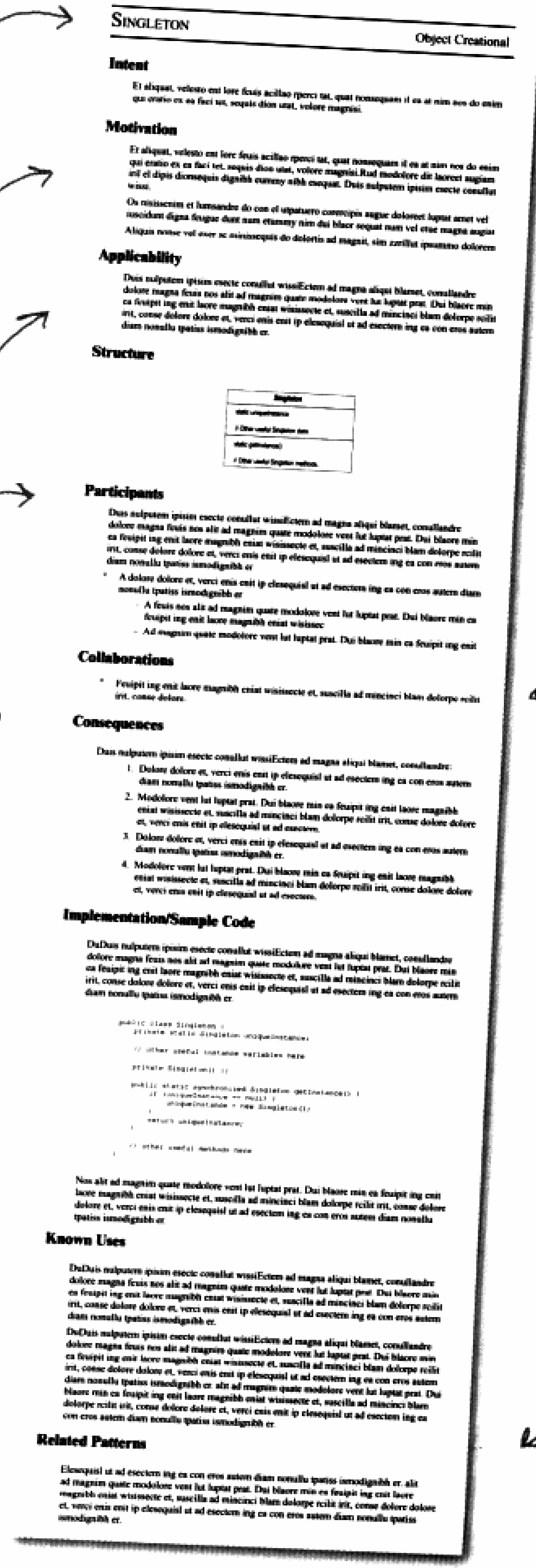
“适用性”描述模式可以被应用在什么场合。

“参与者”描述在此设计中所涉及到的类和对象在模式中的责任和角色。

“结果”描述采用此模式之后可能产生的效果：好的与不好的。

“实现”提供了你在实现该模式时需要使用的技巧，以及你应该小心面对的问题。

“已知应用”用来描述已经在真实系统中发现的模式例子。



这是模式的分类或类目，再过几页我们就会谈到。

“意图”简短地描述该模式的作用。你也可以把它看作是模式的定义（就如同本书中的模式定义一样）。

“结构”提供了图示，显示出参与此模式的类之间的关系。

“协作”告诉我们参与者如何在此模式中合作。

“范例代码”提供代码的片段，可能对你的实现有所帮助。

“相关模式”描述了此模式和其他模式之间的关系。

there are no Dumb Questions

问： 有可能创建自己的设计模式吗？还是只有“模式大师”才办得到？

答： 首先，请务必牢记在心，模式是被“发现的”，而不是被创建的。所以，任何人都可能发现某个设计模式，然后写出它的描述；然而，这并非唾手可得的事情，也不常发生。想成为一个“模式作家”，是需要全力以赴的。

你应该先想想看为何你想发现自己的模式——大多数的人都不是“编写”模式，只是使用模式。然而，你可能是在某一个特定的领域中工作，而你认为新的模式将大有帮助，或者是你找到一个解决方案，能够解决一个再三出现的问题，或者，你只是想要加入设计模式的社群贡献自己的力量。

问： 我有意愿，我要如何开始？

答： 就像任何原则一样，你知道得越多越好。先研究已经被发现的这些模式，了解它们做了些什么，并弄清楚它们和其他模式之间的关系。这些准备工作非常重要，不但可以让你熟悉模式是如何打造出来的，也可以避免做多余的工作。完成这些准备工作之后，你可以开始将你的模式写在纸上，以便与其他开发人员沟通；我们稍后将针对“如何沟通你的模式”多谈一些。如果你真的非常感兴趣，可以阅读本次Q&A以后的内容。

问： 我怎么知道我是否真的有一个模式？

答： 这个问题很好：除非其他人使用它并且发现它很有用，否则你并不算拥有一个模式。一般来说，必须要通过“三次规则”，才算是一个合格的模式。也就是说，只有在真实的世界中被应用三次以上，才能算是一个模式。

您想要当一个设计模式巨星吗？

那么，听清楚了。

先取得一份模式的类目，

然后花些时间好好地学习它。

当你写下一个正确的描述，

而且有三个开发人员都同意你的看法时，

那么你就成功了。

这一段文字也可以改成“你想要当一个摇滚巨星吗？”

想当一个设计模式作家吗？

做好家庭作业。

在发掘新的模式之前，你必须先精通现有的模式。许多模式看起来像是全新的，但是事实上只是现有模式的变种。通过研究现有的模式，你可以比较容易地识别模式，并且学会将某一模式与其他模式联系起来。

花时间反思与评估。

你的经验（你所遭遇过的问题，以及采取的解决方案）正是模式想法的来源。所以，花时间反思过去的经验，并将它用在以后的新设计上面。请牢记，大多数的模式都是现有模式的变种，而非崭新的模式。而且当你真的找到了好像是新模式的东西时，常常都局限在很窄的适用性中，而不能称得上是一个真正的模式。

将你的想法写在纸上，好让其他人能够理解。

如果其他人不能够使用你所找到的模式，那么这个新模式作用也就不大；你需要将你的“准模式”写成一份文档，好让其他人能够阅读、理解，并采用它来解决他们自己的问题，然后将使用的心得反馈给你。很幸运的是，你不需要发明自己的模式归档方法，你可以直接采用四人组的模板。

让其他人使用你的模式，然后再持续改进。

不要认为你可以一次就把模式搞定，应该要把模式当成是随着时间不断进步的一项工程。让其他人评审你的准模式，并尝试着使用它，然后将意见反馈给你。将这些反馈汇总到你的描述中，再重复上述的步骤。你的描述永远不会是完美的，但是到了某个时间点之后，就会相当地稳固，足以让其他开发人员能够阅读并理解它。

不要忘了三次规则。

请记住，除非你的模式已经在真实世界的三个方案中被成功地采用了，否则就不够资格被当成模式。所以，当别人能够　使用你的模式，并将意见反馈给你时，你就有机会能够将它变成一个实用的模式。

使用已有的模式模板定义你的模式，因为这些模板已经包含了许多智慧，而且其他的模式用户也认识这样的格式。



连连看

请将下列模式和描述配对：

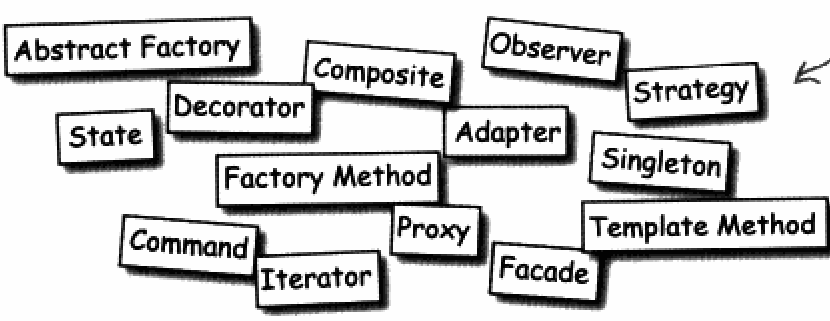
模式	描述
装饰者	封装对象，并提供不同的接口。
状态	由子类决定如何实现一个算法中的步骤。
迭代器	由子类决定要创建的具体类是哪一个。
外观	确保有且只有一个对象被创建。
策略	封装可以互换的行为，并使用委托来决定要使用哪一个。
代理	客户用一致的方式处理对象集合和单个对象。
工厂方法	封装了基于状态的行为，并使用委托在行为之间切换。
适配器	在对象的集合之中游走，而不暴露集合的实现。
观察者	简化一群类的接口。
模板方法	包装一个对象，以提供新的行为。
组合	允许客户创建对象的家族，而无需指定他们的具体类。
单件	让对象能够在状态改变时被通知。
抽象工厂	包装对象，以控制对此对象的访问。
命令	封装请求成为对象。

组织设计模式

随着发掘的设计模式数目逐渐增加，有必要将它们分级分类，好将它们组织起来，以简化我们寻找模式的过程，并让同一群组内的模式互相比较。

在大多数的类目中，模式通常根据某种做法被归为几类。最广为人知的分类方式，就是第一个模式类目中所采用的方式，根据模式的目标分成三个不同类目：创建型、行为型和结构型。

Sharpen your pencil

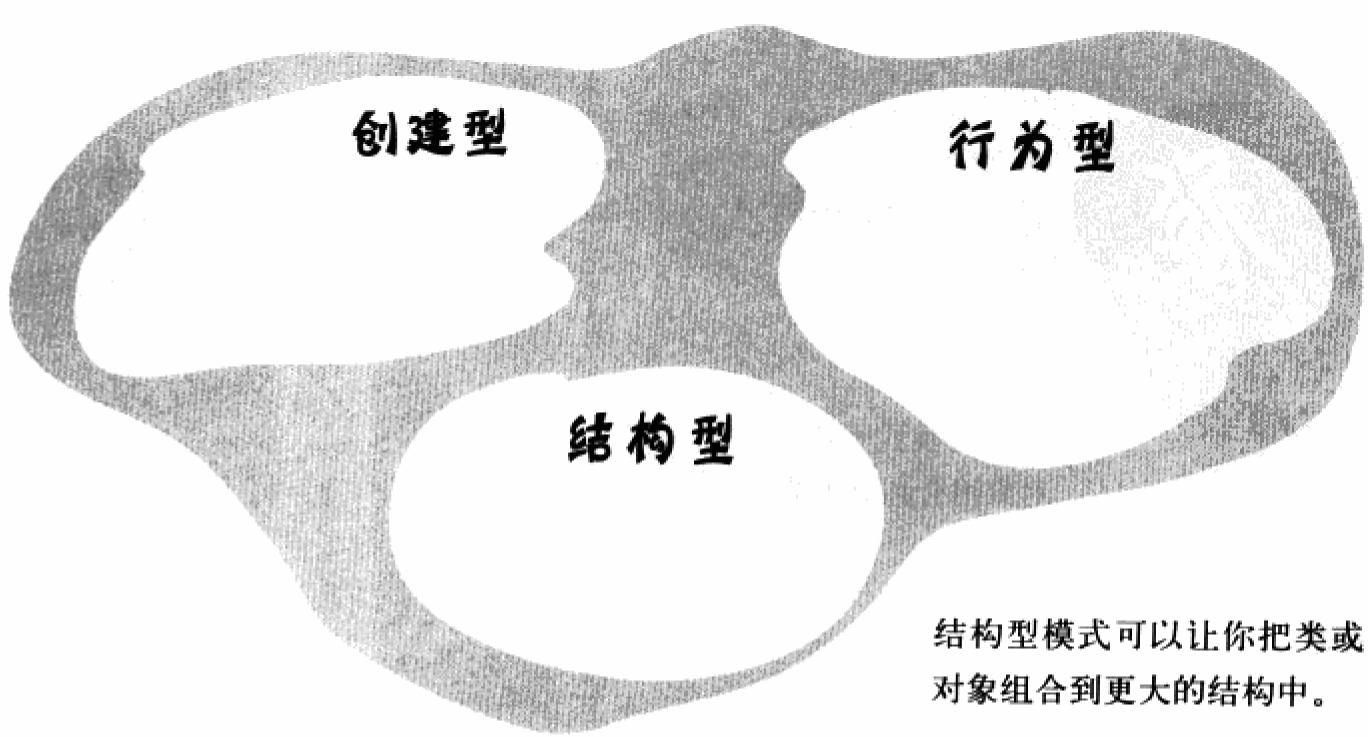


阅读每个类目的描述，试着将这些模式正确地归类。这并不容易！但是请尽力而为。正确答案在下一页。

这里的每个模式都属于以下类目之一。

创建型模式涉及到将对象实例化，这类模式都提供一个方法，将客户从所需要实例化的对象中解耦。

只要是行为型模式，都涉及到类和对象如何交互及分配职责。



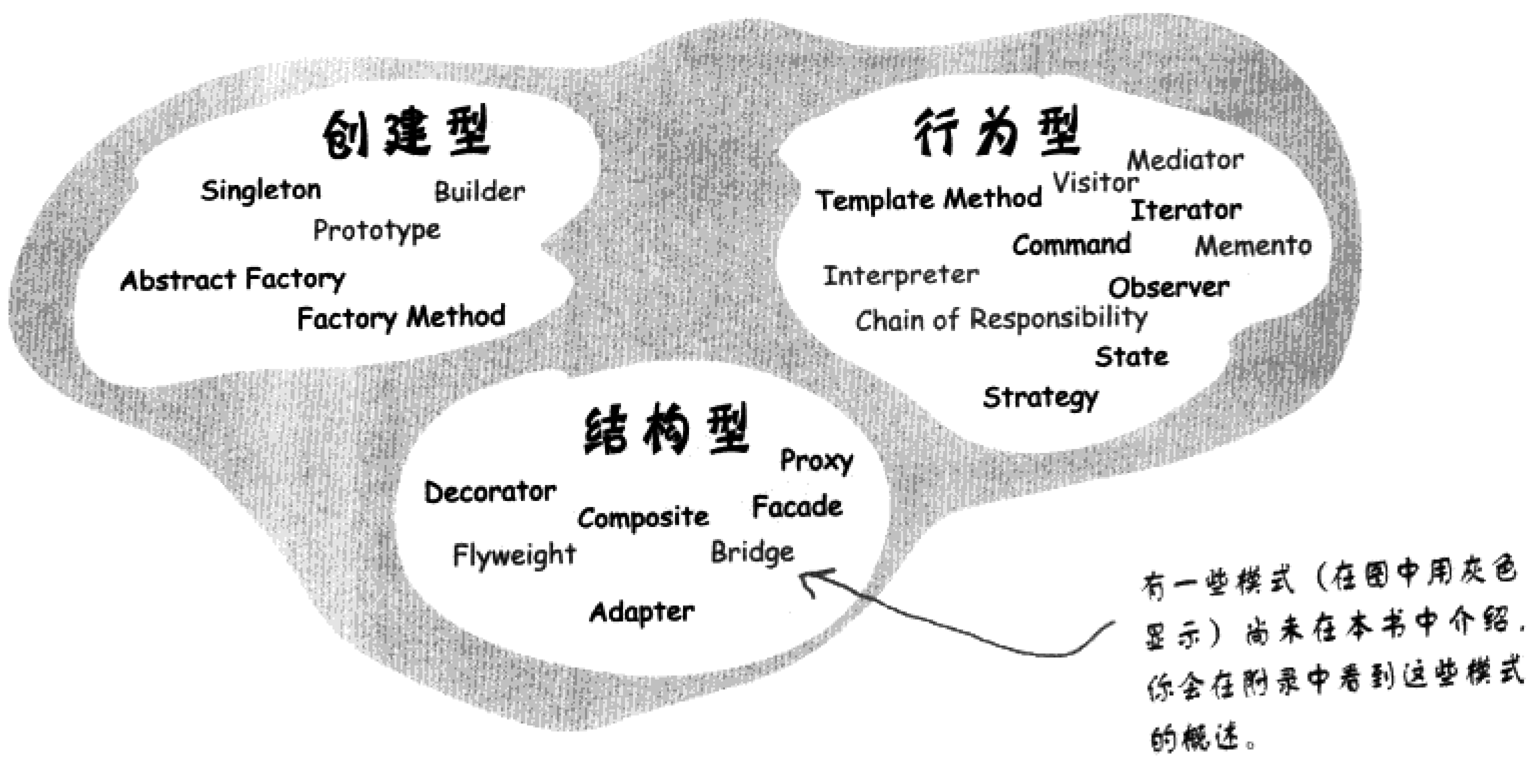
结构型模式可以让你把类或对象组合到更大的结构中。

解答：模式分类

这是把模式分组到类目的结果，你可能觉得这个练习很困难，因为许多模式似乎不只符合一个类目。别担心，其实每个人都有这样的困扰。

创建型模式涉及到将对象实例化，这类模式都提供一个方法，将客户从所需要实例化的对象中解耦。

只要是行为型模式，都涉及到类和对象如何交互及分配职责。

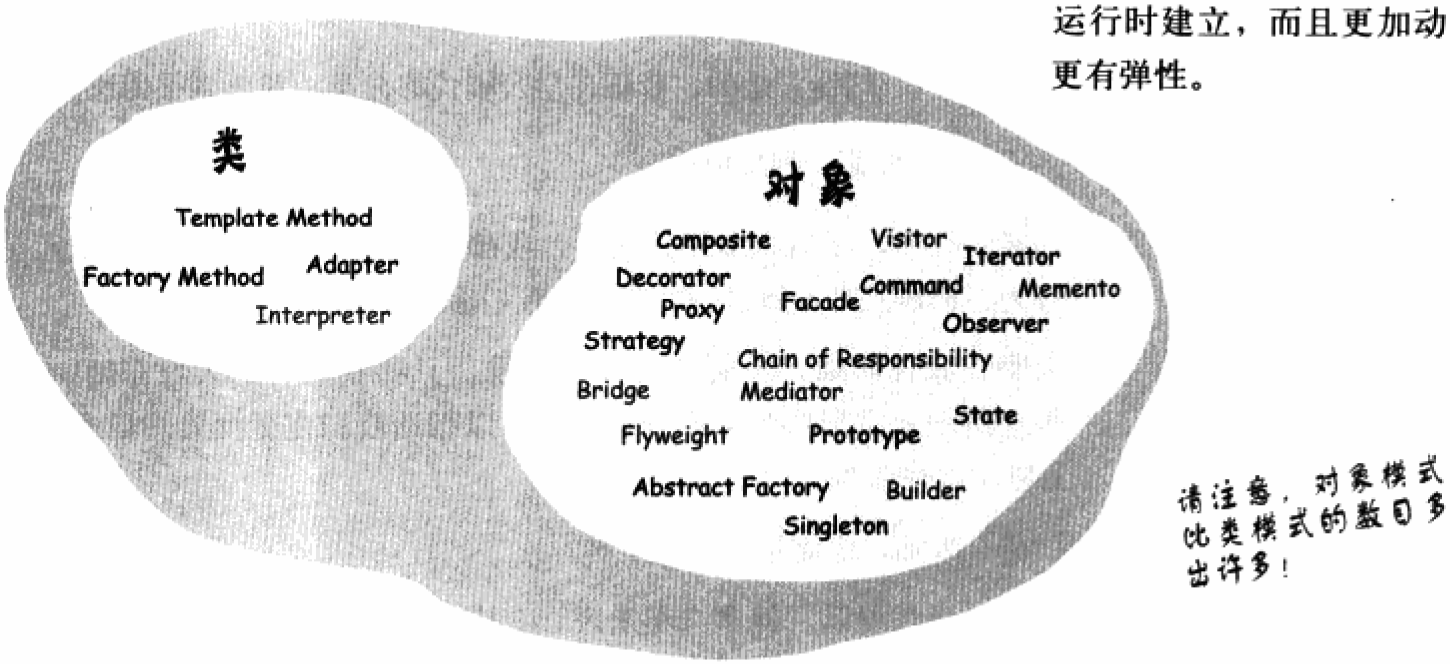


结构型模式可以让你把类或对象组合到更大的结构中。

除了刚才的分类方式之外，模式还有另一种分类方式：模式所处理的是类或对象。

类模式描述类之间的关系如何通过继承定义。类模式的关系是在编译时建立的。

对象模式描述对象之间的关系，而且主要是利用组合定义。对象模式的关系通常在运行时建立，而且更加动态、更有弹性。



there are no Dumb Questions

问：只有这些分类方式吗？

答： 不，还有人提出其他的分类方式。有些分类方式先分成三大类，然后再分成几个小类（例如“解耦模式”）。你想熟悉的是最常用的分类方式。如果建立自己的分类方式可以帮你更加了解这些模式的话，那么你也可以这么做。

问：将模式分成不同的类目，这么做真的能够帮助我们记忆这些模式吗？

答： 通过比较可让你对模式有清晰的概念，这是毋庸置疑的。但是许多人被创建型、结构型和行为型类目搞得一头雾水，常常发现某个模式似乎不只适合一个类目。请记住，怎么分类并不重要，重要的是了解这些模式和它们之间的关系。只要类目有帮助，我们就用它，反之就不用。

问：为何装饰者模式被归类到结构类目中？我认为它应该是行为类目，毕竟它增加行为！

答： 是的，有许多开发人员都这么说！四人组之所以这么分类，他们的想法是这样的：结构型模式用来描述类和对象如何被组合以建立新的结构或新的功能。装饰者模式允许你通过“将某对象包装进另一个对象的方式”，来组合对象以提供新的功能。所以焦点是在于如何动态地组合对象以获取功能，而不是行为型模式的目的——对象之间的沟通与互连。请牢记，这几个模式的意图并不相同，而这通常是了解某个模式属于哪个类目时的关键。



大师与门徒……

大师：蚱蜢啊！你看起来很苦恼。

门徒：是的，我正在学习模式的分类。我感到很困惑。

大师：继续说……

门徒：在学习了这么多模式之后，我被告知每个模式都属于结构、行为、创建三种类目之一。为什么我们需要为模式分类呢？

大师：我说蚱蜢，不管是在什么时候，只要我们有一大堆东西，很自然地就会想要为它们分类，这可以帮助我们在更抽象的层次上思考这些东西。

门徒：大师，你能举一个例子吗？

大师：当然可以。就拿汽车来说，有许多种不同的汽车款式，我们很自然地把汽车分成几类，例如：经济车、跑车、旅行车、卡车及豪华轿车。

大师：蚱蜢，你看起来好像大吃了一惊，难道你无法体会我说的话？

门徒：大师，我很能体会你说的话，我只是对于你如此地了解汽车而感到震惊！

大师：蚱蜢，毕竟不是所有的例子都适合使用莲花或饭钵来举例。现在，我能继续说吗？

门徒：是的，是的，很抱歉打断你，请继续。

大师：一旦你有了分类或类目，你就可以很方便地这么说："如果你想从硅谷开车到圣克鲁斯，那么跑车将会是最好的选择。"或者"因为石油的市场状况日益恶化，所以应该购买经济车，比较省油。"

门徒：所以通过分类，我们可以将一组模式视为一个群体。当我们需要一个创建型模式，但又不知道确切是哪一个的时候，就可以用创建型模式这个词来统称它。

大师：是的，而且分类也有助于我们比较相同类目内的其他成员，比方说，“迷你车是最有风格的小型车。”或者帮助我们缩小搜寻范围，“我需要一部省油的车子。”

门徒：我明白了，所以我就可以说“对于改变对象接口来说，适配器模式是最好的结构型模式”。

大师：是的，类目还可以开发新领域；比方说，“我们真的想要开发一部跑车，具有法拉利的性能和Miata的价格”。

门徒：这种车听起来就像是死亡陷阱。

大师：对不起，我没听清楚你说什么。

门徒：唔！我是说“我懂了”。

门徒：所以类目可以让我们思考模式群组之间的关系，以及同一组模式内模式之间的关系，还可以让我们找出新的模式。但是，为什么使用三个类目，而不是四个或五个？

大师：就像是夜晚天空中的星星一样，你可以看见许多类目。“三”是一个适当的数目，并且是由许多人所决定出来的数目，它有助于更好地进行模式分类。但是的确有人建议用四个、五个或更多个。



用模式思考

情境、约束、力、类目、分类……我的天，听起来非常的学术呢！好吧，这一切都很重要，而知识就是力量。

但是，让我们来面对它，如果你了解理论性的东西，而没有使用模式的经验和实践，那么这将不会在你的生活中造成多大的差别。

下面是一份快速指南，可以帮助你开始“用模式思考”。所谓“用模式思考”，意思是说，能够看着设计，体会在什么地方模式能自然适用，在什么地方模式则不能。



你思考模式的大脑

保持简单（Keep It Simple/KISS）

首先，当你设计时，尽可能地用最简单的方式解决问题。你的目标应该是简单，而不是“如何在这个问题中应用模式”。千万不要认为：如果没有使用模式解决某个问题，就不是经验丰富的开发人员。如果你能够保持简单的设计，那么你将会得到其他开发人员的欣赏和尊敬。正确的说法是，为了要让你的设计简单且有弹性，有时候使用模式是最好的方法。

设计模式非万灵丹；事实上，连什么丹都算不上！

如你所知道的，模式是解决一再发生的问题的通用方案。模式已经被许多开发人员实际测试过。所以，当你需要某个模式的时候，可以放心地使用它，毕竟你知道这个模式已经身经百战。

然而，模式并非万灵丹，你不能把模式插入、编译，然后就早早地去吃午餐。要使用模式，你需要考虑到模式对你的设计中其他部分所造成的后果。

你知道何时需要模式……

啊……这是最重要的问题：何时使用模式？当你在设计的时候，如果确定在你的设计中可以利用某个模式解决某个问题，那么就使用这个模式！如果有更简单的解决方案，那么在决定使用模式之前应该先考虑这个方案。

如何知道何时适用一个模式，这就需要经验和知识。一旦你确定一个简单的解决方案无法满足你的需要，应该考虑这个问题以及相关的约束——这可以帮你将问题对应到一个模式中。如果你对于模式有很深的认知，就可能知道有什么模式适合这样的情况。否则，就花些时间调查一下可能会解决这个问题的模式，模式类目中的意图和应用部分会特别有用。一旦找到了一个看起来适合的模式，要先确定你是否能接受

这个模式所带来的后果，以及对设计其他部分的影响。如果一切看起来都很好，就用它吧！

有一种情况，即使有更简单的解决方案，你仍然想要使用模式，这种情况就是：你预期系统在未来会发生改变。正如我们所见过的，找出你的设计中会改变的区域，通常这是需要模式的迹象。但是务必要确定一件事：加入模式是要应对可能发生的实际改变，而不是假想的改变。

并非只有在设计时才考虑引进模式，在重构（refactoring）时也要这样做！

重构的时间就是模式的时间！

重构就是通过改变你的代码来改进它的组织方式的过程。目标是要改善其结构，而不是其行为。这是一个很好的时机，可以重新检查你的设计来看看是否能够利用模式让它拥有更好的结构。比方说，代码内如果充满了条件语句，这可能意味着需要使用状态模式，或者意味着，应该利用工厂模式将这些具体的依赖消除掉。许多书都介绍在如何利用模式进行重构，而随着技艺的增长，你需要更多地涉猎这个领域。

拿掉你所不需要的，不要害怕将一个设计模式从你的设计中删除。



还没有人谈到何时应该将某个模式删除，你可能认为这很难启齿！不，我们都是成人了，应该面对这个问题。

那么何时应该删除个模式呢？当你的系统变得非常复杂，而且并不需要预留任何弹性的时候，就不要使用模式。换句话说，也就是当一个较简单的解决方案比使用模式更恰当的时候。

如果你现在不需要，就别做。

设计模式威力很强大，你很容易就可以在当前设计中看到模式的各种应用方式。开发人员天生就热爱创建漂亮的架构以应对任何方向的改变。

要抗拒这样的诱惑呀！如果你今天在设计中有实际的需要去支持改变，就放手采用模式处理这个改变吧！然而，如果说理由只是假想的，就不要添加这个模式，因为这只会将你的系统越搞越复杂，而且很可能你永远都不会需要它！



大师与门徒……

大师：蚱蜢，你的基础训练几乎完成了，接下来的计划是什么？

门徒：我要去迪士尼乐园大玩特玩！然后开始利用模式建立许多代码！

大师：等等！你可别忘了“杀鸡焉用宰牛刀”的道理呀！

门徒：这是什么意思呢，大师？我已经学了这么多的设计模式，难道不应该将它们用在我全部的设计中，以达到最强的威力、弹性以及可控性吗？

大师：不，模式只是一种工具，只有在需要时才使用这种工具。你也花了很多时间学习设计原则。一开始总是先遵循这些原则，建立最简单的代码以完成工作。在这个过程中，你看到有需要模式的地方，就使用模式。

门徒：也就是说，我的设计并不是从模式开始？

大师：“应用模式”绝对不是你开始设计时所该有的目标，应该让模式在你的设计过程中自然而然地出现。

门徒：既然模式这么好，为什么在使用它们的时候还得如此小心？

大师：模式可能带来复杂性，如果没有必要，我们绝不需要这样的复杂性。就像你已经知道的，模式是一种被证实过的设计经验，可以避免某些常见的错误。模式也是一种共享的词汇，能够让我们和其他开发人员沟通我们的设计。

门徒：那么，我们又如何知道何时应该引进设计模式呢？

大师：当你确信你的设计中有一个问题需要解决的时候，或者当你确信未来的需求可能会改变时，都可以采用模式。

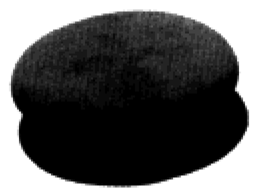
门徒：虽然我已经了解了许多的模式，但我觉得我的学习应该继续下去。

大师：是的，蚱蜢。学习管理软件的复杂度和变化，这是一生的课题。但是现在既然你已经知道了许多模式，就可以开始在需要的地方采用它们，并不断地学习更多的模式。

门徒：等一下，你是说我还没有学完“全部”？

大师：蚱蜢，你已经学会了一些基础模式，你会发现还有更多的模式在等着你，包括一些应用在特定领域的模式，例如并发系统（Concurrent System）和企业系统。现在你已经有了良好的基础，学习这些模式就不会太难！

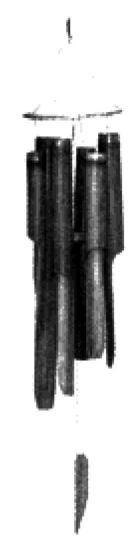
使用模式的心智



初学者的心智

"我要为Hello World找个模式"

初学者到处使用模式。这很好：初学者可以借此培养许多使用模式的实战经验。初学者也认为"我使用越多模式，我的设计就越好"。初学者将慢慢认识到并非如此，所有的设计都应该尽量保持简单。只有在需要实践扩展的地方，才值得使用复杂性和模式。



随着学习的进程，中级人员的心智开始能够分辨何时需要模式，而何时不需要。中级人员的心智依然会企图把过多的模式套用在不适当的地方，但他们也开始察觉到有些模式并不适合目前的情况，可以对其改编使其适合。

中级人员的心智

"或许这里我需要一个单件模式。"



悟道者的心智

"在这里采用装饰者模式相当自然。"

悟道者的心智能够看到模式在何处能够自然融入。悟道者的心智并不急切于使用模式，而是致力于最能解决问题的简单方案。悟道者的心智会考虑对象的原则，以及它们之间的折衷。当对模式的需要自然出现时，悟道者的心智就拿捏得宜地采用模式。悟道者的心智也能看到相似模式之间的关系，以及它们在意图上的微妙差异。悟道者的心智也同于初学者的心智——不会让这些模式的知识过度影响设计的决策。

警告：过度使用设计模式可能导致代码被过度工程化。应该总是用最简单的解决方案完成工作，并在真正需要模式的地方才使用它。



当然我们希望你使用设计模式！

但是我们更希望你能够成为一个好的面向对象设计者。

当一个设计方案决定要使用某个模式的时候，将为你带来好处，因为任何模式都是身经百战，被验证了是能够解决其问题的。而且模式可以被良好地归档，容易被其他开发人员所了解（你知道的，模式是开发人员共享的词汇）。

然而，当你使用设计模式的时候，仍然会有缺点。设计模式常常产生一些额外的类和对象，所以会增加设计的复杂度。设计模式也会在你的设计中加入更多层，这不但增加复杂性，而且效率下降。

另外，有时使用设计模式会大材小用。许多时候回头看看设计原则，你会发现有简单得多的解决方案能解决相同的问题。若果真如此，可别抗拒，就用较简单的解决方案吧！

不要因我们的话而感到挫折。我们并非鼓励你不要用模式。当设计模式应用得恰当时，好处其实是非常多的。

别忘了共享词汇的威力

在这本书中，我们花了相当多的时间讨论面向对象的基础知识，但可别忘了设计模式中人的一面——设计模式不仅可以帮助你在大脑中装进这些解决方案，也可以让你和其他开发人员之间有共享的词汇，而这正是设计模式最大的优点之一。

想想看，从上次我们谈到共享词汇至今，有些事情已经不一样了；你现在已经开始建立了自己的某些词汇！更别说学会了一整套的面向对象设计原则，而从这些设计原则中你能轻易了解所遇到的任何新模式的动机和工作方式。

现在你已经有了设计模式的基础，应该“把模式传出去”，让大家都知道。为什么呢？因为当你的同伴开发人员也知道这些模式并使用共享词汇的时候，将使得你们的设计更好，更容易沟通。最棒的是，你省下了大量的时间。



共享词汇的五种方式

1. **在设计会议中**：当你和你的团队在会议中讨论软件设计时，使用设计模式可以帮你们待在“设计中”久一点。从设计模式和面向对象原则的视角讨论设计，可以避免你的团队很快地陷入实现的细节，也可以避免发生许多误解。

2. **和其他开发人员**：当你和其他开发人员讨论的时候，可以使用模式。这可以帮助其他开发人员学习新模式，并建立一个社群。和别人分享你所学会的东西是很有成就感的一件事情。

3. **在架构文档中**：当你在编写架构文档的时候，使用模式将会缩减文档的篇幅，并且让读者更清楚地了解你的设计。

4. **在代码注释以及命名习惯上**：当你在编写代码的时候，应在注释中清楚地注明你所使用的模式。在选择类和方法的名称时，应尽可能显示出隐藏在下面的模式。其他的开发人员在阅读你的代码时会感激你，因为你让他们能够很快地了解你的实现。

5. **将志同道合的开发人员集合在一起**：分享你的知识。许多开发人员都听说过模式，但并不真正了解什么是模式。你可以自愿为他们讲一堂模式介绍课，或者成立一个读书会。



和四人组一同巡游对象村

在对象村内，你不会遇到“喷射帮”和“鲨鱼帮”（译注：电影“西城故事”（West Side Story）中的两个帮派），但是你有机会遇到四人组。你大概也注意到了，想要在模式的世界中走得够远，你就一定会遇到他们。那么，到底这个神秘的“帮派”是怎么一回事呢？

简单地说，四人组包括了Erich Gamma、RichardHelm、Ralph Johnson和John Vlissides。他们是第一群将模式归类的功臣，而这个过程开启了软件领域的一大跃进。

这个称号又是怎么来的？没有人知道，反正大家都是这么称呼。但是想想看：如果你想成为“帮派成员”，好好地认识对象村，那么该怎么办呢？幸好，他们同意带我们一同去巡游对象村……

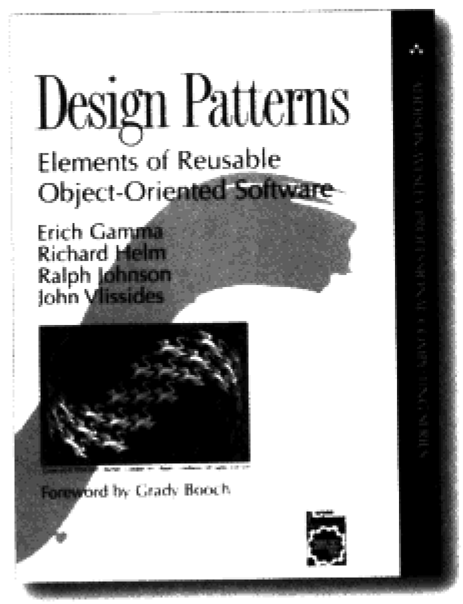
四人组发起了软件模式运动，随后有许多人也做出了重大的贡献，包括Ward Cunningham、Kent Beck、Jim Coplien、Grady Booch、Bruce Anderson、Richard Gabriel、Doug Lea、Peter Coad和Doug Schmidt，上面只列出了一小部分名单。





你的旅途刚刚开始……

现在你已经站在设计模式的顶端，准备挖得更深了；我们为你准备了比较权威的三本书，把它们添加到你的书架上吧……



设计模式的经典书籍

这本书在1995年出版，揭开了设计模式的序幕。你可以在这本书中找到所有基础的模式。事实上，这本书中所介绍的模式，也正是本书的基础。

这本书并非涵盖了所有的模式——从这本书出版之后，这个领域就不断地扩大——但尽管如此，它还是第一本也是最重要的一本书。

在你读完《Heed First设计模式》之后，拿起这本书来探索模式是个很棒的选择。

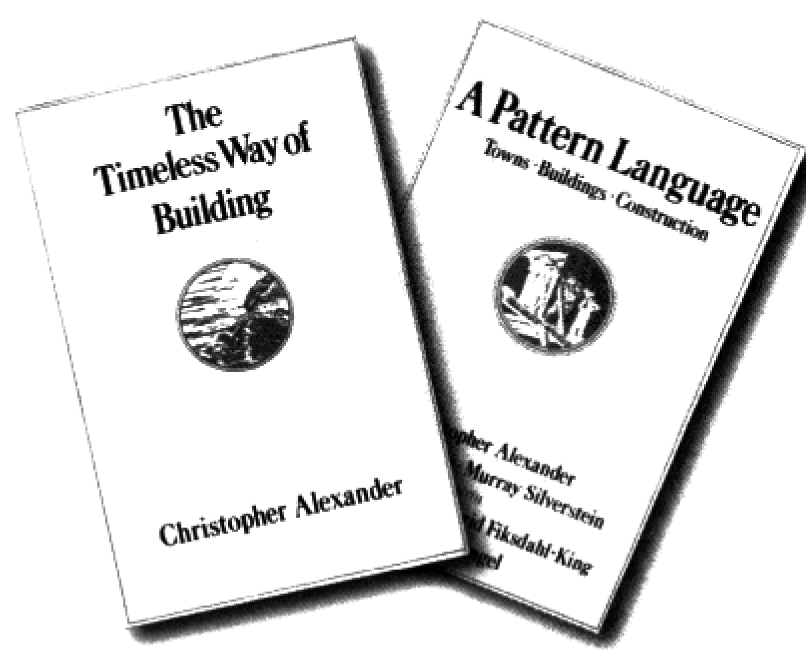
这本书的作者后来被称为"四人组"，或简称GoF。

模式的经典书籍

模式并不是从四人组开始的，而是始于ChristopherAlexander。他是伯克利的建筑学教授——没错，Alexander是个建筑师，而不是计算机科学家。Alexander发明了建筑模式（像房屋、城镇和城市）。

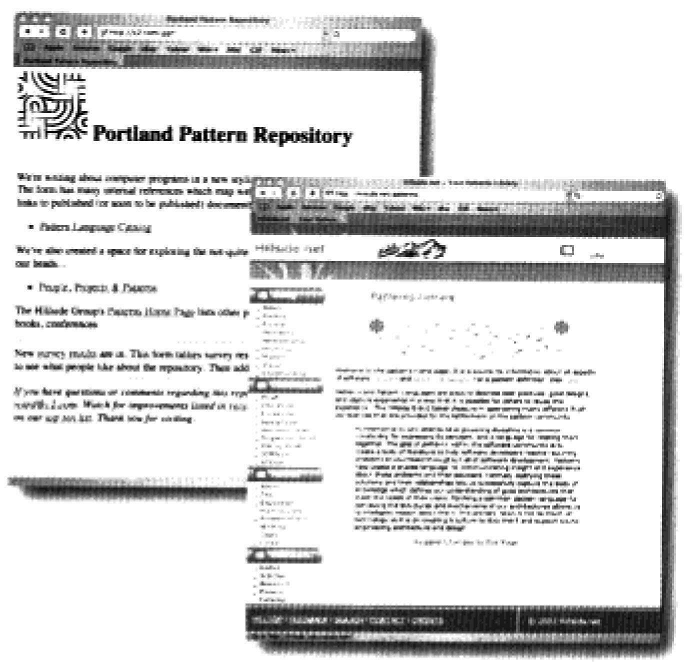
下次当你有心情想挖掘得更深入时，可以阅读《The Timeless Way of Building》和《A Pattern Language》这两本书，从中你会了解到设计模式的真正起源，并体会到创建"有生命的"建筑和具有弹性、可扩展性软件之间的对比。所以，拿起你的星巴兹咖啡，坐下靠在椅背上，开始享受这一切吧……

Christopher Alexander发明了模式，导致软件也产生了类似的解决方案。



其他设计模式资源

你会发现外面有许多活跃、热情的模式使用者和设计者社群，他们正敞开双臂等候你的加入。这里列出一些你一开始可以取得的资源……



网 站

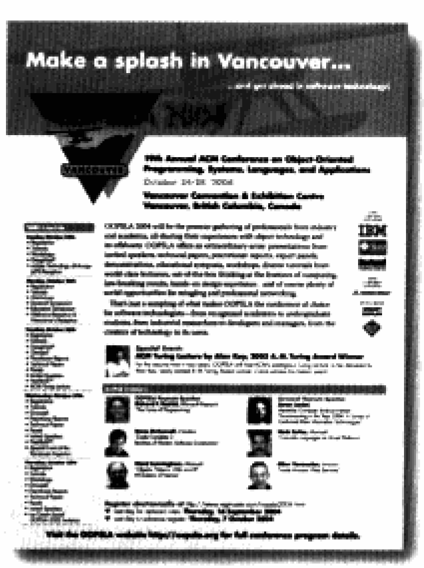
The Portland Patterns Repository，由Ward Cunningham运作，这是一个致力于模式相关信息的WIKI，任何人都可以加入。

你可以看到许多你能想到的有关模式和OO系统的问题在这里都有主题讨论。

http://c2.com/cgi/wiki?WelcomeVisitors

The Hillside Group，旨在促进通用的编程和设计实践，并提供模式的集中资源。这个网站包含了许多模式相关资源的信息，例如文章、书籍、邮件列表和工具。

http://hillside.net/



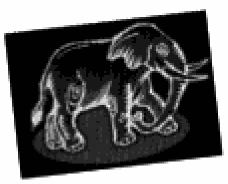
会议和研讨会

如果你想和模式社群面对面地接触，一定要查看有哪些与模式相关的会议和研讨会。Hillside网站有完整的清单。另外你至少也应该去看看OOPSLA的活动信息。OOPSLA是ACM举办的研讨会，主题是针对面向对象系统、语言和应用。

模式动物园



就如我们刚刚所说的，模式并非从软件开始，而是始于建筑和城镇的架构。事实上，模式的概念可以被应用在许多不同的领域。现在就让我们来逛逛模式动物园，瞧瞧有哪些模式……

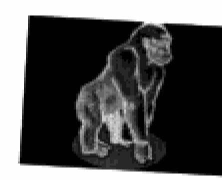


架构模式

用来建立生气勃勃的建筑、城镇和城市的架构。这也正是模式开始的地方。

栖息地：从你所居住、观看、参观的建筑物中，可以发现它的踪迹。

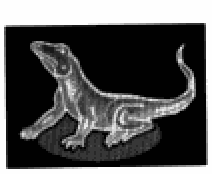
应用模式



是建立系统级架构的模式。许多多层的架构都属于这一类目。

栖息地：出现在三层架构、客户/服务器系统以及Web中。

野外笔记：MVC可算是其中的一种。



领域特定模式

关注特定领域的问题，例如并发系统或实时系统。

帮忙找到它的栖息地

J2EE

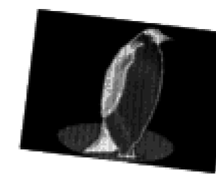
业务流程模式

描述业务、顾客和数据之间的交互，此种模式能够处理如“如何有效决策并沟通决策”之类的问题。



出没在公司的会议室以及项目管理会议中。

组织模式



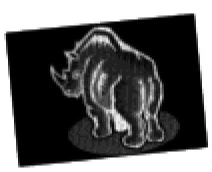
描述了人类组织的结构以及实践。到目前为止大多数努力聚焦于制造或支持软件的组织。

帮助找到一个栖息地

开发团队

顾客支持团队

用户界面设计模式

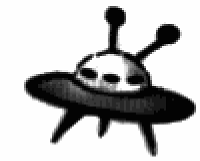


致力于解决设计交互式软件时的问题。

栖息地：被发现在视频游戏设计者、GUI构造者和制作者附近。

野外笔记：请将你对模式领域的观察和发现写在这里。

以反模式歼灭恶势力



如果我们只有模式，而没有反模式，那么这个宇宙就不完整了。如果设计模式能够让你在某个特定的情境之下，对一再出现的问题提供通用的解决方案，那么反模式能给你什么？

> **反模式**告诉你如何采用一个不好的解决方案解决一个问题。

反模式看起来总像是一个好的解决方案，但是当它真正被采用后，就会带来麻烦。

通过将反模式归档，我们能够帮助其他人在实现它们之前，分辨出不好的解决方案。

像模式一样，有许多类型的反模式，包括了开发反模式、OO反模式、组织反模式和领域特定反模式。

你可能会这么问："怎么会有人愿意浪费时间将不好的解决方案归档？"

这么说好了：如果老是有人用某个不好的解决方案处理某个问题，而通过将它归档，可以帮助其他开发人员避免犯同样的错误。毕竟，避免不好的解决方案，就和发现好的解决方案一样有价值！

让我们来看看一个反模式的元素：

反模式告诉我们为什么不好的解决方案会有吸引力。

必须面对的是，如果不好的解决方案没有任何吸引力，那么根本就不会有人想要使用它。反模式最重要的工作之一，在于警告你不要陷入某种致命的诱惑。

反模式告诉你为何这个解决方案从长远看会造成不好的影响。

为了了解为什么这是一个反模式，你必须了解它在将来如何造成负面影响。反模式会告诉你使用这个解决方案，在将来会为你带来怎样的麻烦。

反模式建议你改用其他的模式以提供更好的解决方案。

反模式除了告诉你什么解决方案不好之外，也会为你指出正确的方向，向你建议一些会引向好的解决方案的可能性，这样反模式才真正有帮助。

现在就让我们来看一个反模式。

下面是一个软件开发反模式的例子。

像设计模式一样，每个反模式都有名字，所以我们可以创建共享词汇。

反模式

名称：黄金榔头

问题与情境，如同设计模式所使用的描述。

问题：你需要为你的开发选择技术，而且你相信正好有一种技术能够主宰这个架构。

情境：你需要开发某个新的系统或者是一套软件，然而此系统或软件却无法和开发团队所熟悉的技术相吻合。

告诉你为什么这个解决方案是有吸引力的。

力：

- 开发团队致力于采用他们所熟悉的技术。
- 开发团队并不熟悉其他技术。
- 采用不熟悉的技术被认为风险比较高。
- 使用熟悉的技术做开发，比较容易规划和预估。

不好的但有吸引力的解决方案。

原本的解决方案：反正就使用熟悉的技术好了。将熟悉的技术强迫性地用在许多问题上，甚至在明显不适当的地方也照用。

如何使用一个好的解决方案。

重构的解决方案：开发人员通过教育、培训和读书会，可以学会新的解决方案。

这个反模式会出现在什么地方。

例子：

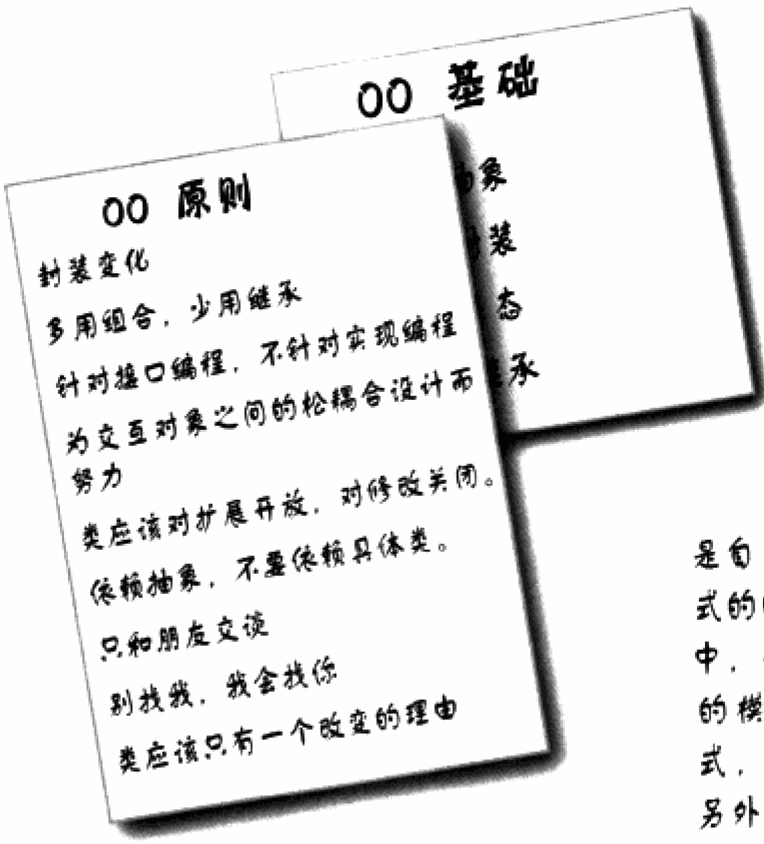
当采用开放源码的替代品时，Web公司依然持续使用并维护他们内部自行开发的缓存系统。

此例取自Portland Pattern Repository的Wiki，网址是http://c2.com/，在那里你可以发现许多反模式和相关的讨论。



设计箱内的工具

你已经到了可以脱离我们的阶段，现在该是你走向外面的世界，凭着自己的能力探索模式的时候了。



是自己走出去发现更多模式的时候了。外面的世界中，有许多属于特定领域的模式和一些基础的模式，并未在本书中提及。另外，也可以创建你自己的模式。

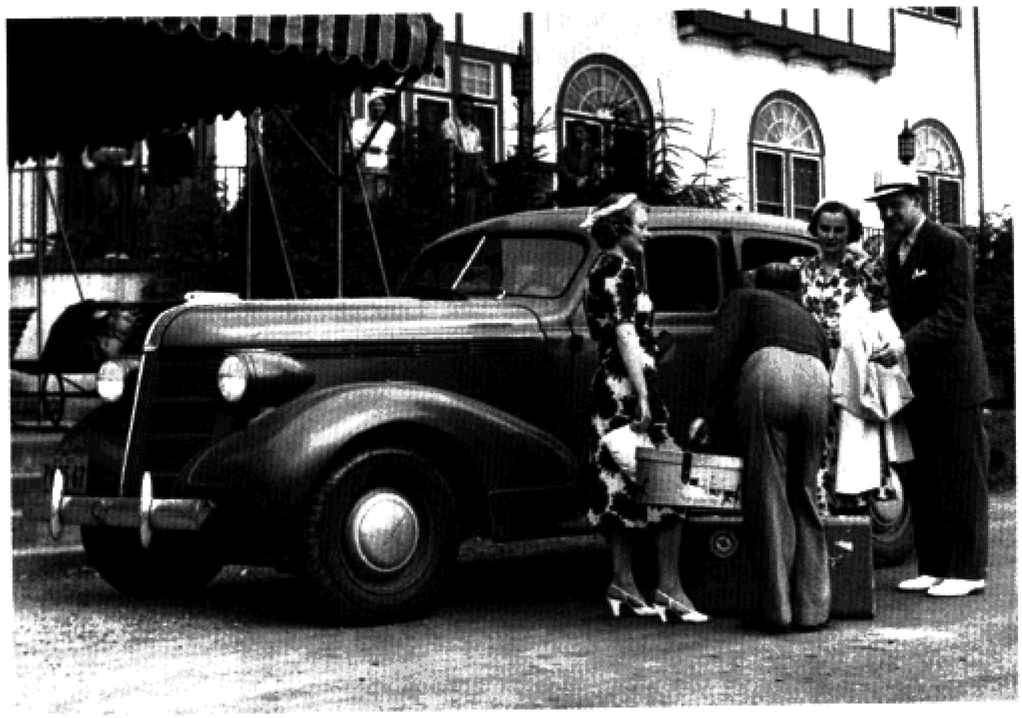


去看看附录，有一些更基础的模式，你可能会感兴趣。

要点

- 让设计模式自然而然地出现在你的设计中，而不是为了使用而使用。
- 设计模式并非僵化的教条；你可以依据自己的需要采用或调整。
- 总是使用满足需要的最简单解决方案，不管它用不用模式。
- 学习设计模式的类目，可以帮你自己熟悉这些模式以及它们之间的关系。
- 模式的分类（或类目）是将模式分成不同的族群，如果这么做对你有帮助，就采用吧!
- 你必须相当专注才能够成为一个模式的作家：这需要时间也需要耐心，同时还必须乐意做大量的精化工作。
- 请牢记：你所遇到大多数的模式都是现有模式的变体，而非新的模式。
- 模式能够为你带来的最大好处之一是：让你的团队拥有共享词汇。
- 任何社群都有自己的行话，模式社群也是如此。别让这些行话绊着，在读完这本书之后，你已经能够应用大部分的行话了。

离开对象村……



有你们的日子真好。

我们一定会想念你们的。但是，别担心，下一本Head First书很快就会出版，到时候欢迎你们再度来访。你问我下一本书是什么主题？这……真是好问题！你要不要给点意见？发E-mail到booksuggestions@wickedlysmart.com吧！

习题解答

连连看

请将下列模式和描述配对：

模式

- 装饰者
- 状态
- 迭代器
- 外观
- 策略
- 代理
- 工厂方法
- 适配器
- 观察者
- 模板方法
- 组合
- 单件
- 抽象工厂
- 命令

描述

- 封装对象，并提供不同的接口。
- 由子类决定如何实现一个算法中的步骤。
- 由子类决定要创建的具体类是哪一个。
- 确保有且只有一个对象被创建。
- 封装可以互换的行为，并使用委托来决定要使用哪一个。
- 客户用一致的方式处理对象集合和单个对象。
- 封装了基于状态的行为，并使用委托在行为之间切换。
- 在对象的集合之中游走，而不暴露集合的实现。
- 简化一群类的接口。
- 包装一个对象，以提供新的行为。
- 允许客户创建对象的家族，而无需指定他们的具体类。
- 让对象能够在状态改变时被通知。
- 包装对象，以控制对此对象的访问。
- 封装请求成为对象。

附录A

剩下的模式



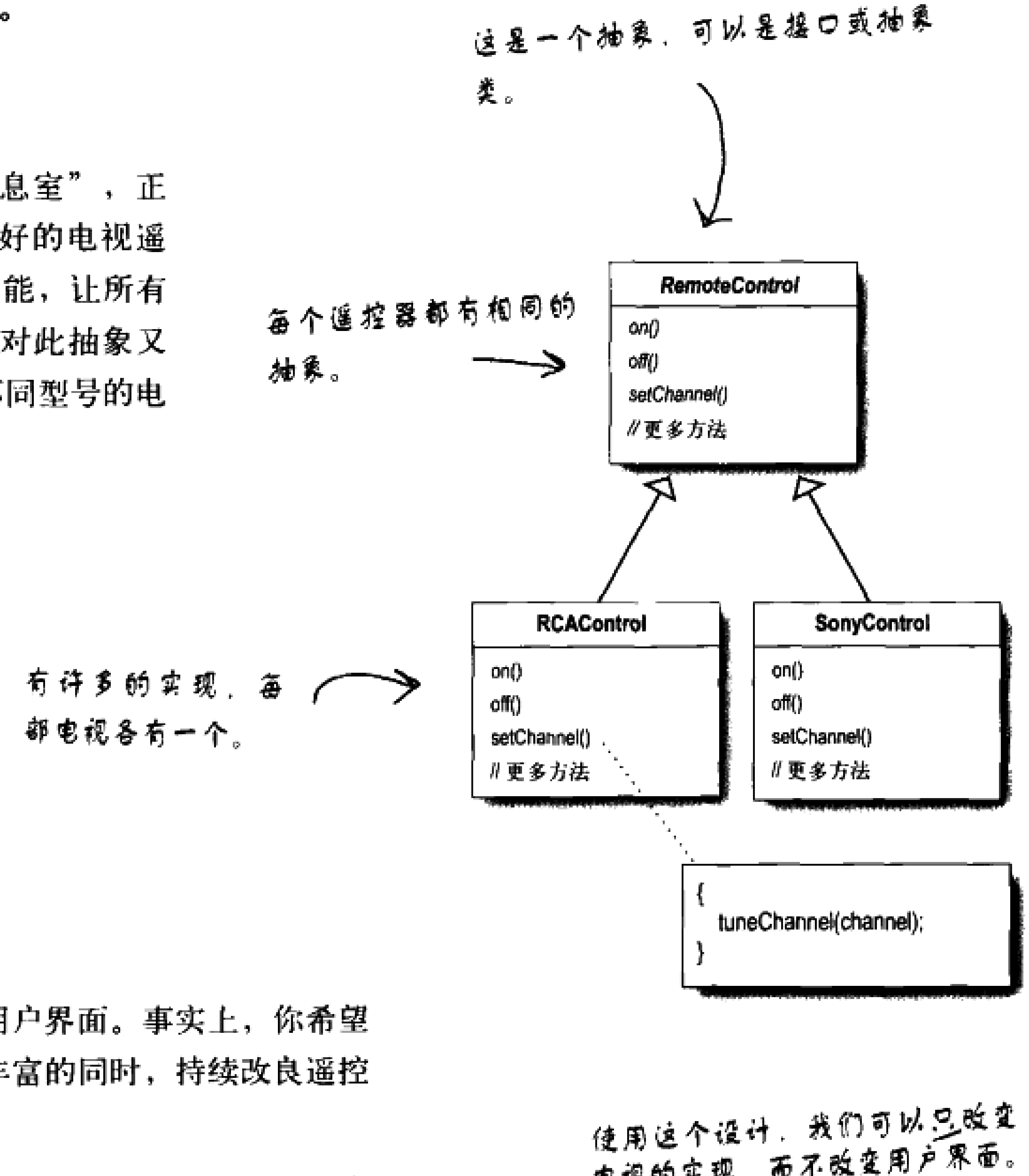
并非每个人都广受欢迎。 过去10年来，事情改变了许多。自从《设计模式：可复用面向对象软件的基础》一书出版之后，开发人员就开始大量地采用这些模式。我们在此附录中所介绍的模式，都是成熟、典型、正式的四人组模式，只不过可能不像前面章节所探索的模式那么经常地被使用。但是这些模式本身也有相当可取之处，而如果你遇到了合适的情形，也应当毫不犹豫地采用它们。我们在此的目标，是希望能够让你通盘了解这些模式的意义。

桥接

使用桥接模式（Bridge Pattern）不只改变你的实现，也改变你的抽象。

场景

你打算彻底改革你的“极限休息室”，正为一个新的人体工学且接口友好的电视遥控器编程。你要使用好的OO技能，让所有的遥控器基于相同的抽象，而对此抽象又做出许多不同的实现——每部不同型号的电视都有自己的遥控器实现。



你的两难

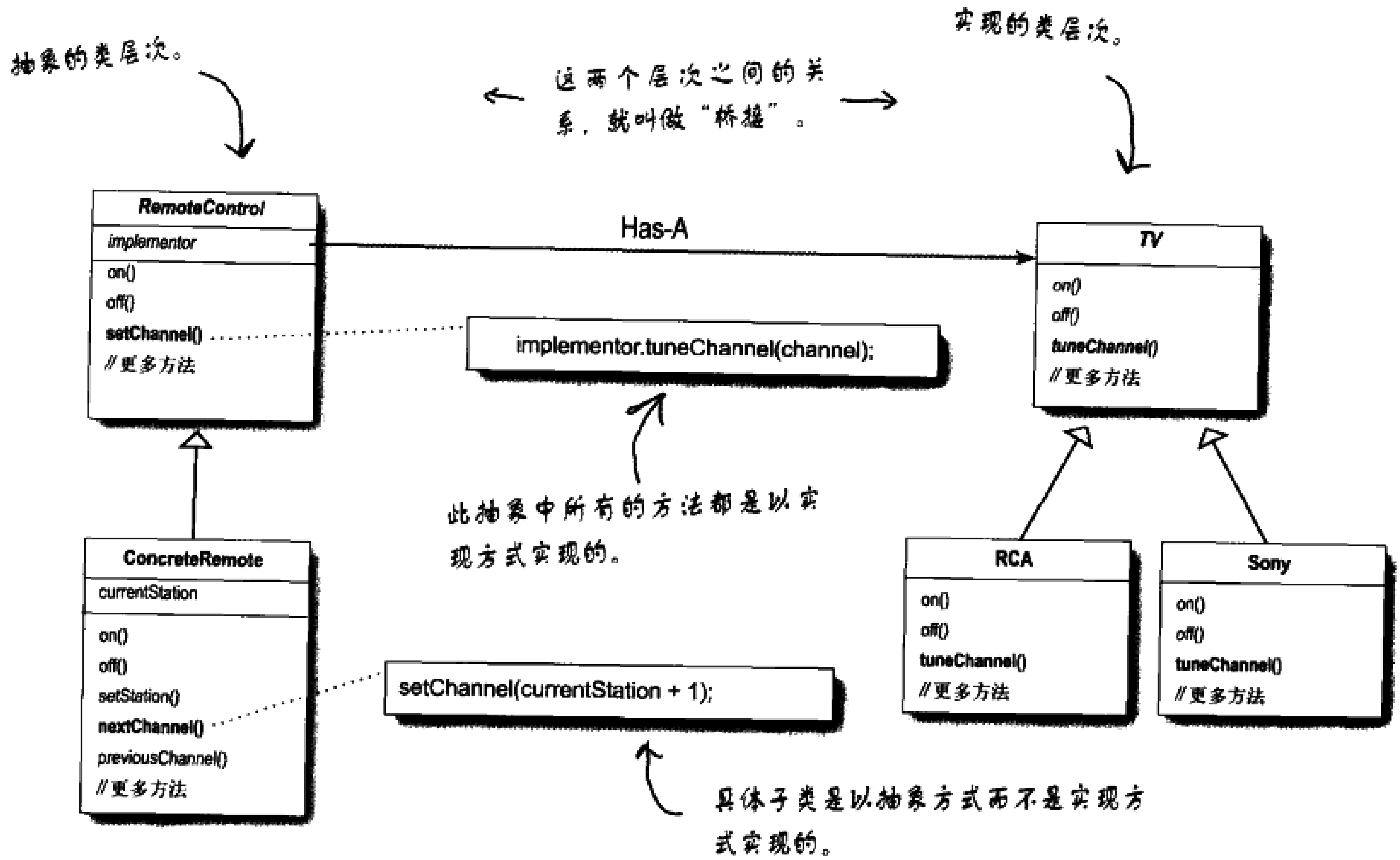
你不会第一次就做对遥控器的用户界面。事实上，你希望随着可用性数据收集得越来越丰富的同时，持续改良遥控器。

所以你的两难之处就在于：遥控器会改变，而电视机也会改变。你已经将用户界面抽象出来，所以可以根据不同的电视机改变它的实现。事情还不只这样，随着使用时间的增长,用户会对此界面提出一些想法，你还必须应对他们的反馈来改变抽象。

所以你要如何建立一个OO设计，能够改变实现和抽象呢？

为何使用桥接模式？

桥接模式通过将实现和抽象放在两个不同的类层次中而使它们可以独立改变。



现在你有了两个层次结构，其中一个是遥控器，而另一个是平台特定的电视机实现。有了桥接的存在，你就可以独立地改变这两个层次。

桥接的优点

- 将实现予以解耦，让它和界面之间不再永久绑定。
- 抽象和实现可以独立扩展，不会影响到对方。
- 对于"具体的抽象类"所做的改变，不会影响到客户。

桥接的用途和缺点

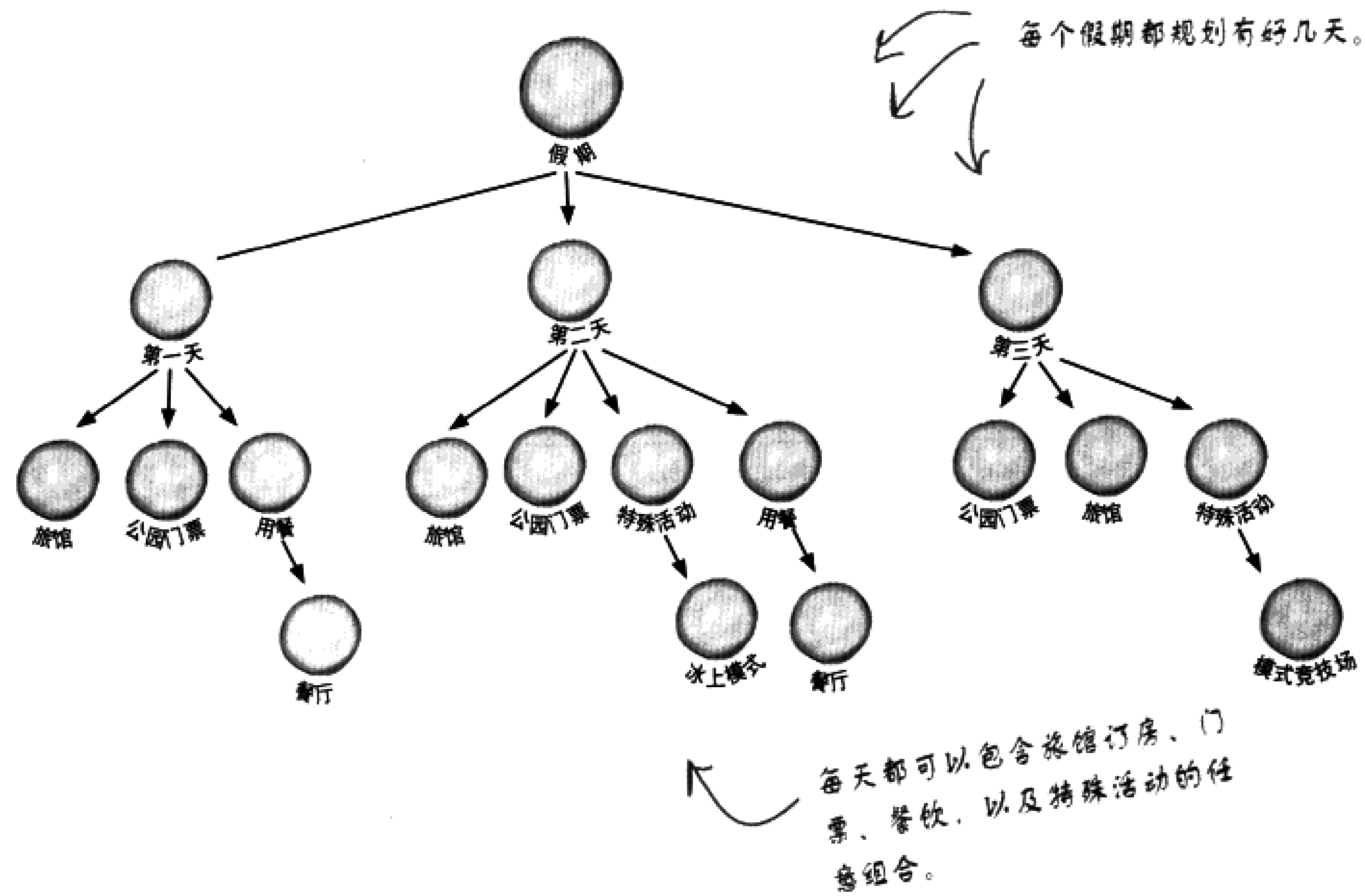
- 适合使用在需要跨越多个平台的图形和窗口系统上。
- 当需要用不同的方式改变接口和实现时，你会发现桥接模式很好用。
- 桥接模式的缺点是增加了复杂度。

生成器

使用生成器模式（Builder Pattern）封装一个产品的构造过程，并允许按步骤构造。

场景

“模式乐园”是在对象村外围的一个新主题公园，他们请你为“模式乐园”制定一套度假计划。客人可以选择旅馆以及各种门票、餐厅订位，甚至也可以选择登记参加特殊的活动。想要制定一套度假计划，你需要建立像下面这样的结构：



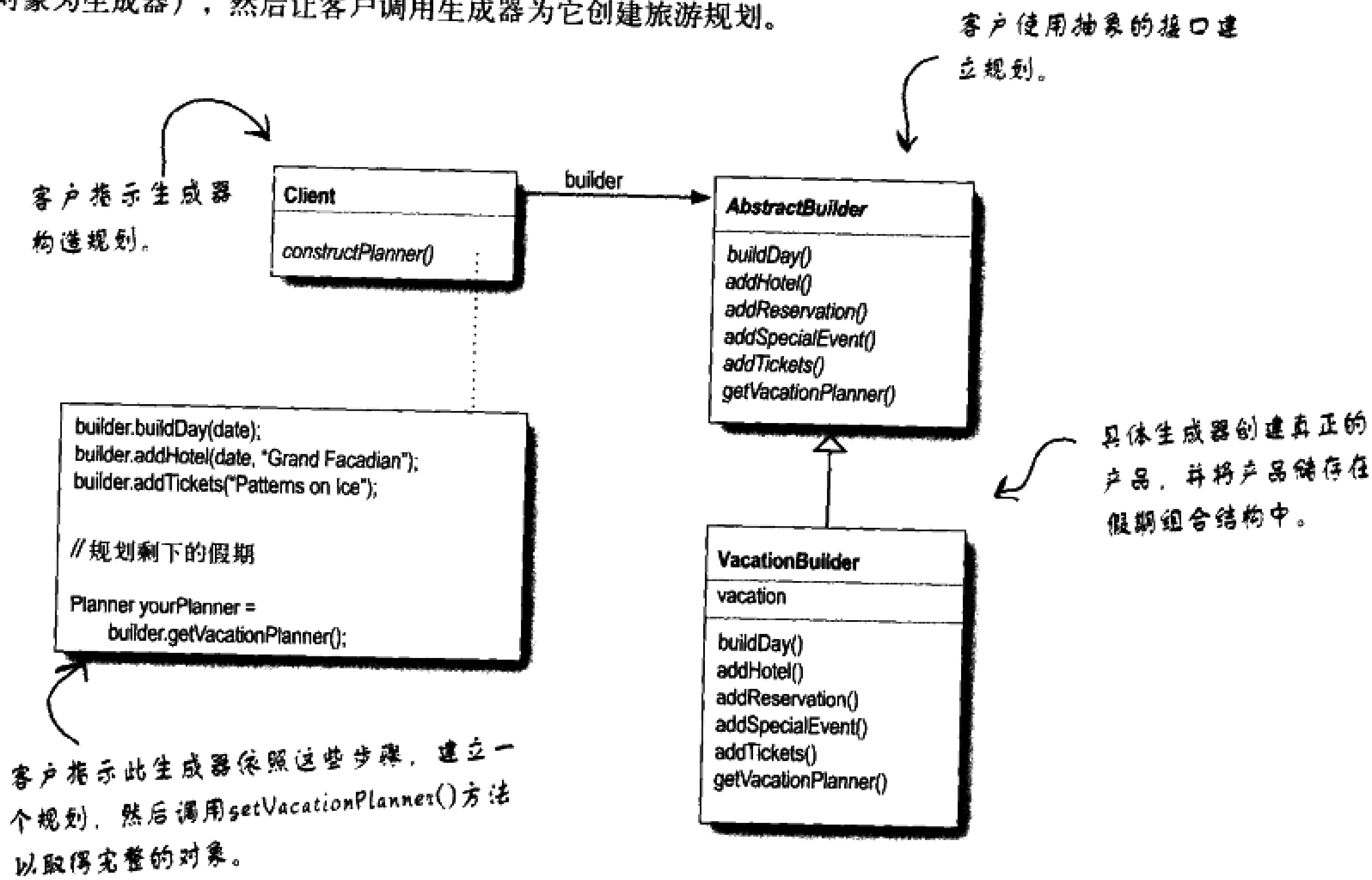
你需要一个有弹性的设计

每个客人的度假计划可能都不太一样，例如天数、活动类型。比方说，当地居民可能不需要旅馆，但是想要用餐并参与特殊活动。而其他的客人可能是从外地飞过来的，所以需要旅馆、用餐和门票。

所以，你需要一个有弹性的数据结构，代表客人的规划，以及所有的变化；你也需要遵照一系列潜在的复杂顺序，创建这样的规划。你要如何才能够提供一种方式来创建这个复杂的结构，而不会和创建它的步骤混在一起呢？

为何使用生成器模式？

还记得迭代器吗？我们将迭代的过程封装进入一个独立的对象中，并向客户隐藏集合的内部表现。这里也是采取相同的想法：我们将旅游规划的创建过程，封装到一个对象中（让我们称此对象为生成器），然后让客户调用生成器为它创建旅游规划。



生成器的优点

- 将一个复杂对象的创建过程封装起来。
- 允许对象通过多个步骤来创建，并且可以改变过程（这和只有一个步骤的工厂模式不同）。
- 向客户隐藏产品内部的表现。
- 产品的实现可以被替换，因为客户只看到一个抽象的接口。

生成器的用途和缺点

- 经常被用来创建组合结构。
- 与工厂模式相比，采用生成器模式创建对象的客户，需要具备更多的领域知识。

责任链

当你想要让一个以上的对象有机会能够处理某个请求的时候，就使用责任链模式（Chain of Responsibility Pattern）。

场景

自从推出Java版本的糖果机之后，万能糖果公司收到的电子邮件数量已超出他们所能处理的范围。据他们自己分析，所收到的电子邮件有四类：其一，Fans寄来的信，他们喜欢新推出的1 in 10游戏；其二，父母寄来的信，抱怨他们的孩子沉溺于这个游戏；其三，店家寄来的信，他们希望能够在某些地方也摆设糖果机；其四，垃圾邮件。

所有Fans的邮件都需要直接送到CEO手上，所有的抱怨邮件则是送给法律部门，而所有的新机器请求邮件则交给业务部门，至于垃圾邮件当然是删除了事。



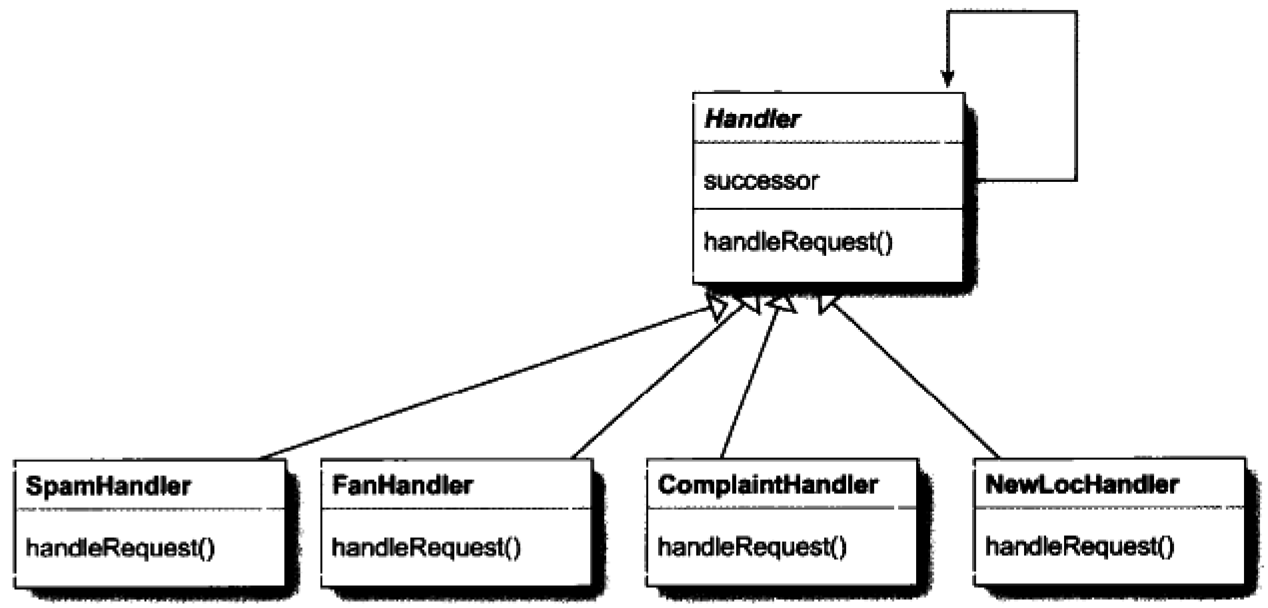
你的任务

万能糖果公司已经写了一些人工智能过滤程序，这些程序很厉害，它们会分辨邮件是属于上述哪一类，但是他们需要你构造一个设计——使用这个过滤程序处理收到的邮件。

如何使用责任链模式

通过责任链模式，你可以为某个请求创建一个对象链。每个对象依序检查此请求，并对其进行处理，或者将它传给链中的下一个对象。

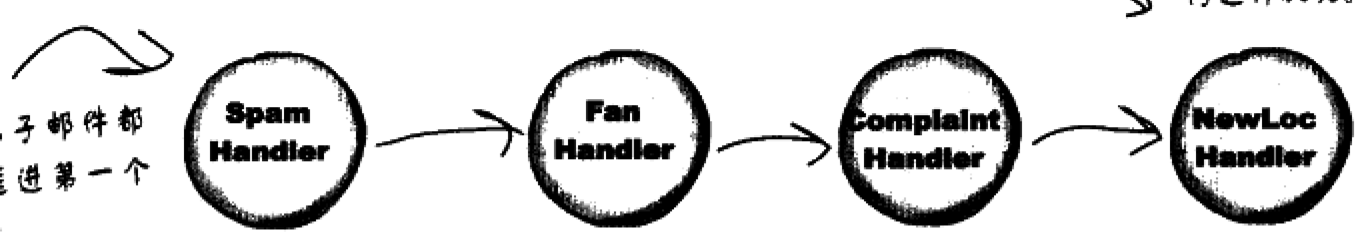
链中的每个对象扮演处理器，并且有一个后继对象。如果它可以处理请求，就进行处理；否则把请求转发给后继者。



当收到电子邮件的时候，它会被送进第一个处理器，也就是SpamHandler。如果SpamHandler无法处理，就将它传给FanHandler。依次类推……

如果电子邮件掉落到链尾端之后，就表示它没有经过任何处理——不过你可以实现一个终极处理器应付这种状况。

每个电子邮件都会被送进第一个处理器。



责任链的优点

- 将请求的发送者和接受者解耦。
- 可以简化你的对象，因为它不需要知道链的结构。
- 通过改变链内的成员或调动它们的次序，允许你动态地新增或者删除责任。

责任链的用途和缺点

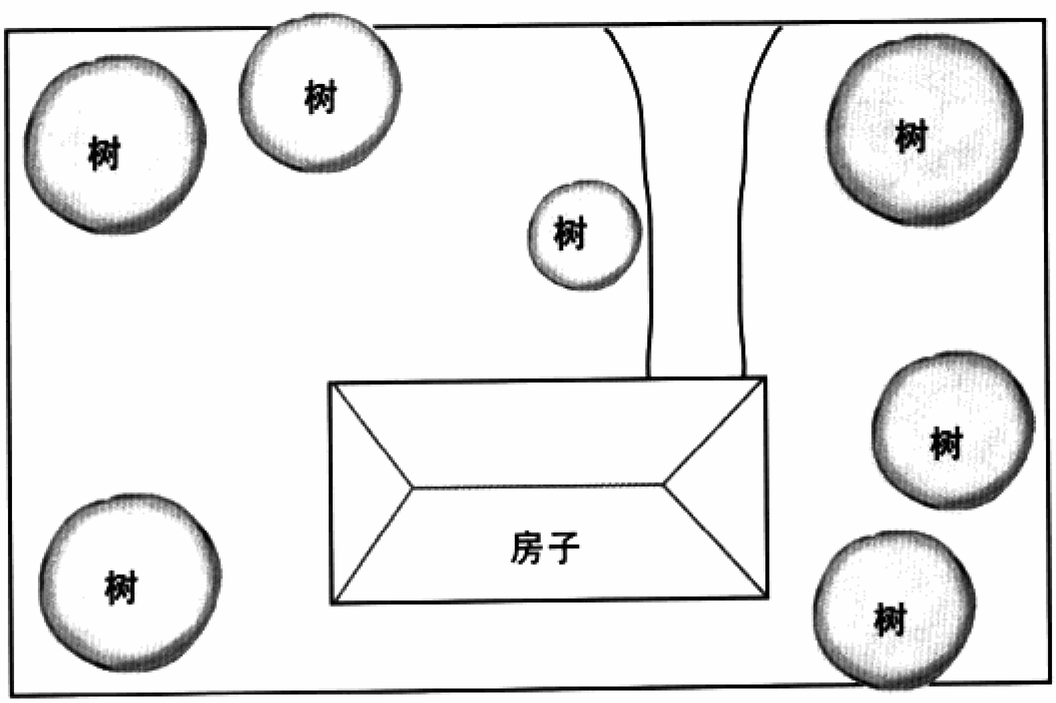
- 经常被使用在窗口系统中，处理鼠标和键盘之类的事件。
- 并不保证请求一定会被执行；如果没有任何对象处理它的话，它可能会落到链尾端之外（这可以是优点也可以是缺点）。
- 可能不容易观察运行时的特征，有碍于除错。

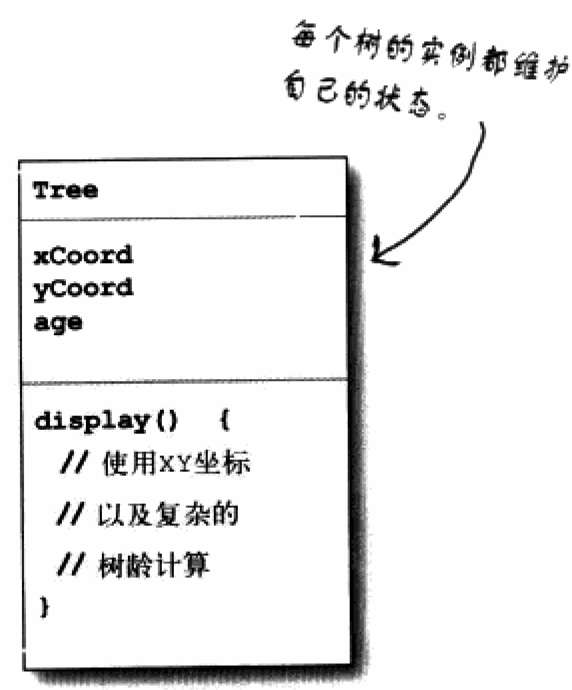
蝇量

如想让某个类的一个实例能用来提供许多"虚拟实例"，就使用蝇量模式（Flyweight Pattern）。

场景

在热门的全新景观设计应用中，你想要加上一些树作为点缀；树有一个XY坐标位置，而且可以根据树的年龄动态地将自己绘制出来。问题是，用户可能要在他们的家庭景观设计中有非常非常多的树，看起来就像这样：



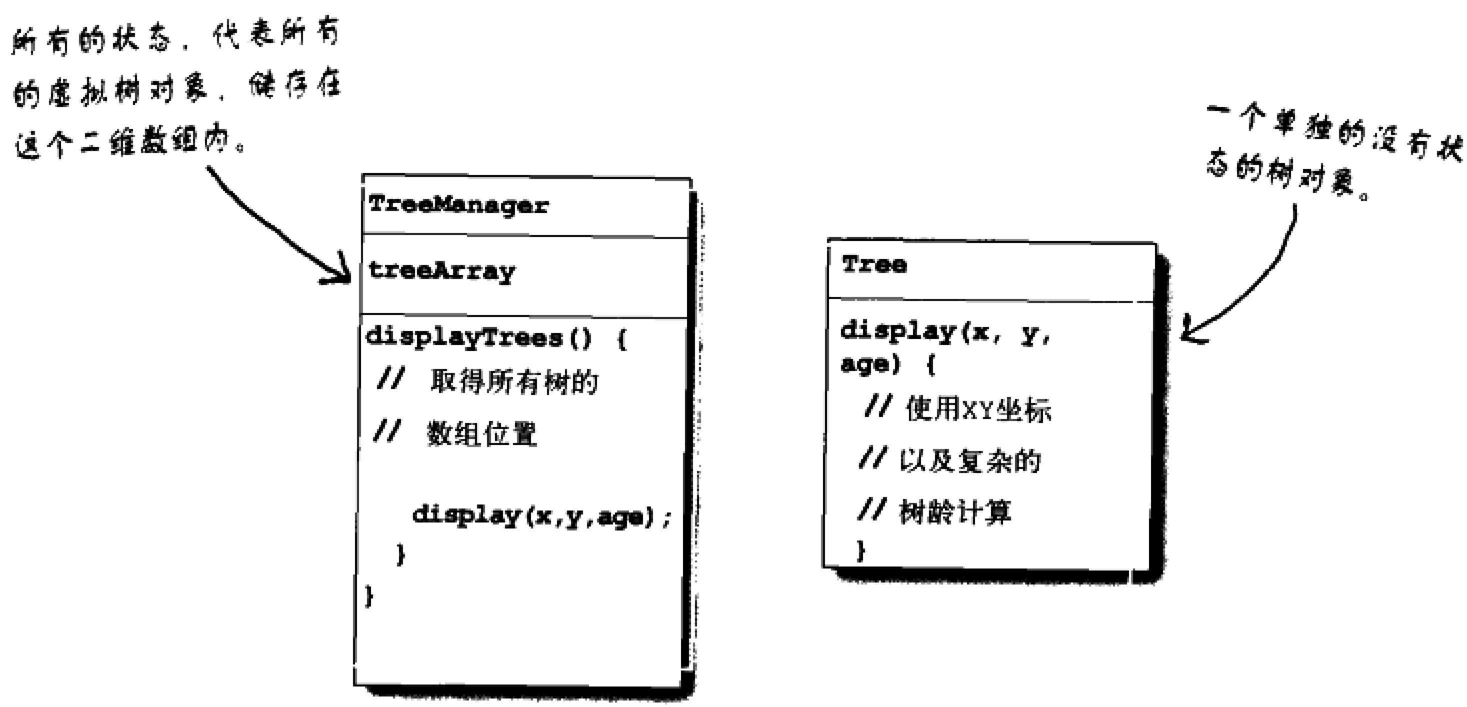


你的大客户陷入两难

你刚刚取得了重大突破。你已经向关键客户努力推销了好几个月，而他们打算购买1,000套你的软件，并将其用于大型规划社区的景观设计。在使用一个星期之后，客户开始抱怨：他们创建了许多树之后，这个程序开始变得呆滞……

为何使用蝇量模式？

如果不用上页的做法，你可以重新设计系统，只用一个树实例和一个客户对象来维护“所有”树的状态。这就是蝇量模式！



蝇量的优点

- 减少运行时对象实例的个数，节省内存。
- 将许多“虚拟”对象的状态集中管理

蝇量的用途和缺点

- 当一个类有许多的实例，而这些实例能被同一方法控制的时候，我们就可以使用蝇量模式。
- 蝇量模式的缺点在于，一旦你实现了它，那么单个的逻辑实例将无法拥有独立而不同的行为。

解释器

使用解释器模式（Interpreter Pattern）为语言创建解释器。

场景

还记得Duck Pond 的模拟器吗？你可能会想到这适合拿来当做儿童学习编程的教育工具。使用这个模拟器，每个孩子都可以用一种简单的语言来控制一只鸭子。下面是此语言的一个简单例子：

```
right;
while (daylight) fly;
quack;
```

让鸭子右转。

整天都在飞翔……

……然后呱呱叫。

现在，回想很久以前，你在编程入门课程上所学到的语法知识，把语法写成下面这样：

```
expression ::=  <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')'<expresion>
variable ::= [A-Z,a-z]+
```

程序是一个表达式，内含一串命令和重复（"while"语句）。

所谓的"一串"，指的是一群表达式，彼此之间用分号隔开。

我们有三个命令：向右、呱呱叫，以及飞行。

while语句由一个条件变量和一个表达式组成。

放轻松

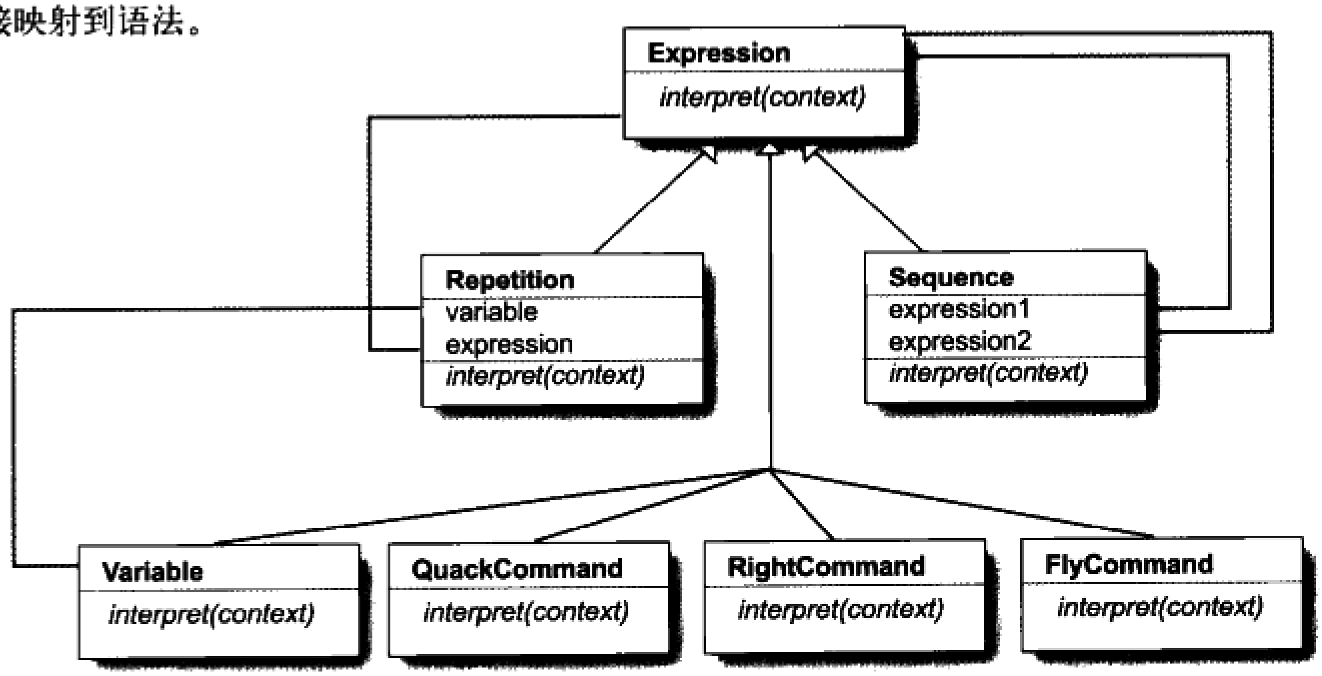
解释器模式需要一些形式语法的知识，如果你还没有学过形式语法，那么请继续读下去，你还是可以抓住一些重点的。

现在怎么办?

你已经有了一个语法；现在所需要做的事情，就是表现并解释语法中的句子，好让学生看到用这个语言控制鸭子的效果。

如何实现解释器

当你需要实现一个简单的语言时，就使用解释器模式定义语法的类，并用一个解释器解释句子。每个语法规则都用一个类代表。这是一个将鸭子语言转化成类的例子，请特别留意，类直接映射到语法。



要想解释这种语言，就调用每个表达式类型的interpret()方法。此方法需要传入一个上下文（Context）——也就是我们正在解析的语言字符串输入流——然后进行比对并采取适当的动作。

解释器模式的优点

- 将每一个语法规则表示成一个类，方便于实现语言。
- 因为语法由许多类表示，所以你可以轻易地改变或扩展此语言。
- 通过在类结构中加入新的方法，可以在解释的同时增加新的行为，例如打印格式的美化或者进行复杂的程序验证。

解释器的用途和缺点

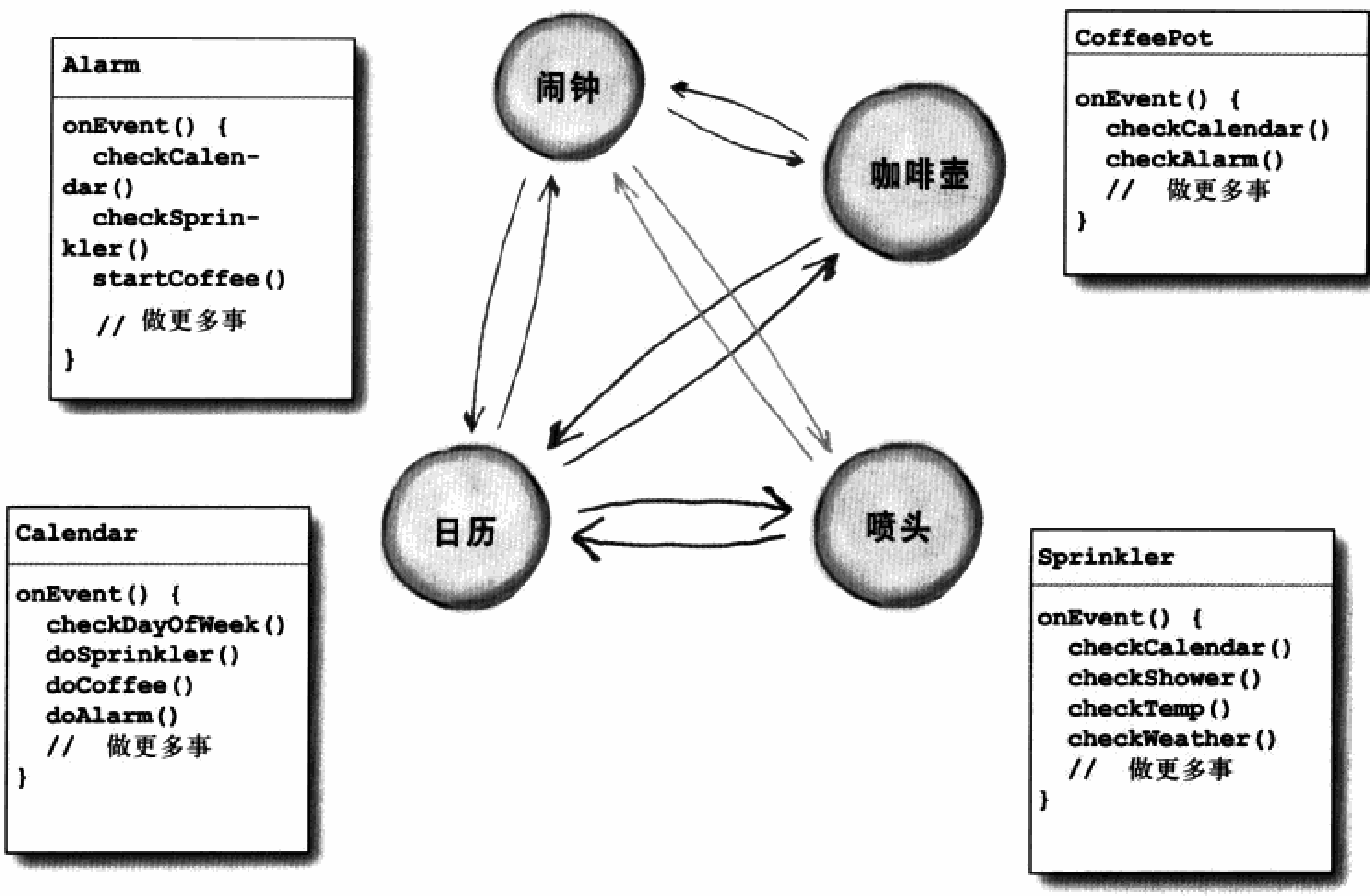
- 当你需要实现一个简单的语言时，使用解释器。
- 当你有一个简单的语法，而且简单比效率更重要时，使用解释器。
- 可以处理脚本语言和编程语言。
- 当语法规则的数目太大时，这个模式可能会变得非常繁杂。在这种情况下，使用解析器/编译器的产生器可能更合适。

中介者

使用中介者模式（Mediator Pattern）来集中相关对象之间复杂的沟通和控制方式。

场景

感谢未来屋公司的这群好家伙，Bob拥有一个Java版本的自动屋，这可以让他的生活变得更便利。当Bob点击了打盹按钮，他的闹钟就会告诉咖啡壶开始煮咖啡。尽管生活对他来说是如此惬意，但他（以及其他的客户）总是不断地提出许多新的要求：周末不要供应咖啡……在洗澡前将喷头关闭15分钟……在丢垃圾的日子里将闹钟时刻提前……



未来屋公司的两难

想要持续地追踪每个对象的每个规则，以及众多对象之间彼此错综复杂的关系，实在不容易。

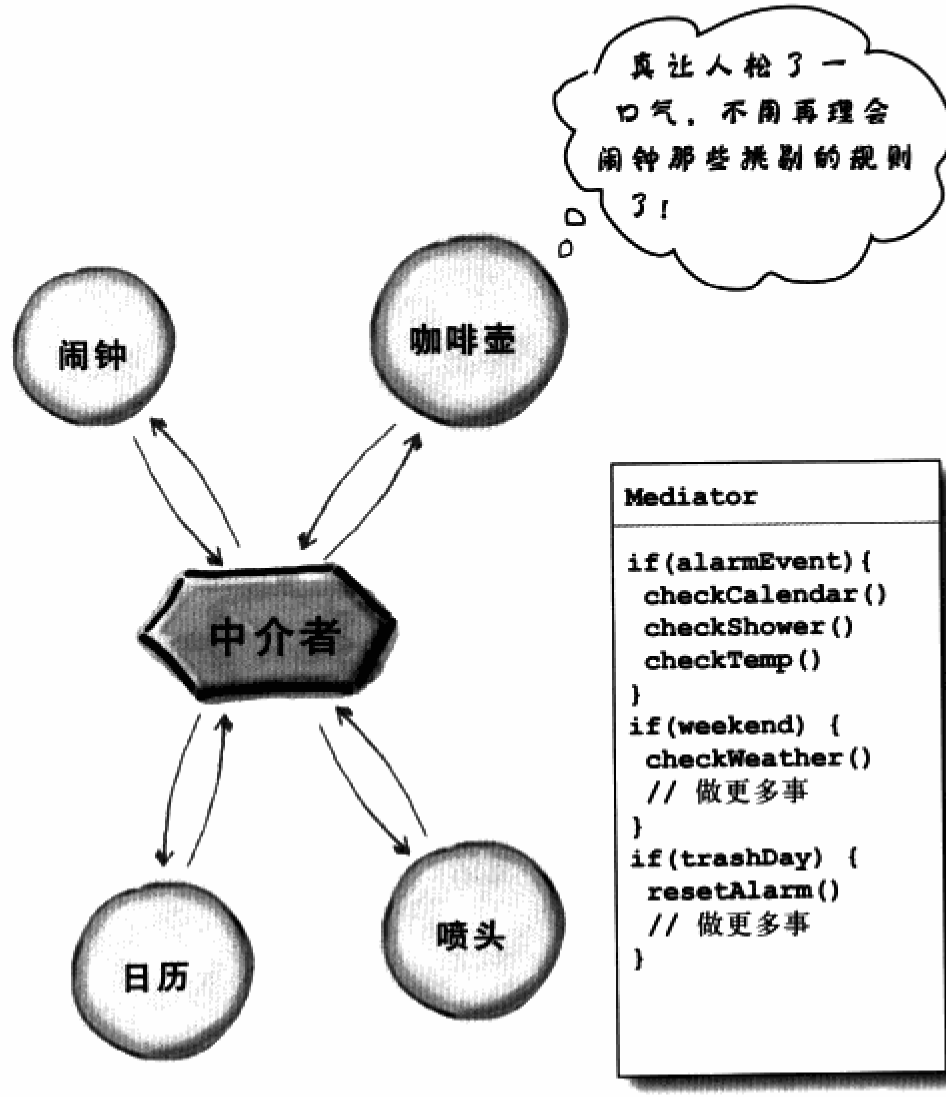
中介者在行动……

在这个系统中加入一个中介者，一切都变得简单了。

- 每个对象都会在自己的状态改变时，告诉中介者。
- 每个对象都会对中介者所发出的请求作出回应。

在没有中介者的情况下，所有的对象都需要认识其他对象……也就是说，对象之间是紧耦合的。有了中介者之后，对象之间彻底解耦。

中介者内包含了整个系统的控制逻辑。当某装置需要一个新的规则时，或者是一个新的装置被加入系统内，其所有需要用到的逻辑也都被加进了中介者内。



中介者的优点

- 通过将对象彼此解耦，可以增加对象的复用性。
- 通过将控制逻辑集中，可以简化系统维护。
- 可以让对象之间所传递的消息变得简单而且大幅减少。

中介者的用途和缺点

- 中介者常常被用来协调相关的GUI组件。
- 中介者模式的缺点是，如果设计不当，中介者对象本身会变得过于复杂。

备忘录

当你需要让对象返回之前的状态时（例如，你的用户请求“撤销”），就使用备忘录模式（MementoPattern）。

场景

你的交互式角色扮演游戏获得了巨大的成功，大家都很沉迷，想要进入“第13关”。当用户进入到更高的游戏关卡时，游戏结束的机率就会提高。对于那些花了许多日子才进入到高级关卡的游戏迷，当他们的角色死在游戏中时，他们简直是气炸了，他们一定会重新再来的。于是他们强烈要求你提供“储存进度”的命令，好让玩家能够储存游戏进度，至少不要损失得太严重。这个“储存进度”的功能需要设计成能够抛出一个复活的角色，而进度停留在上一次过关的关卡上。

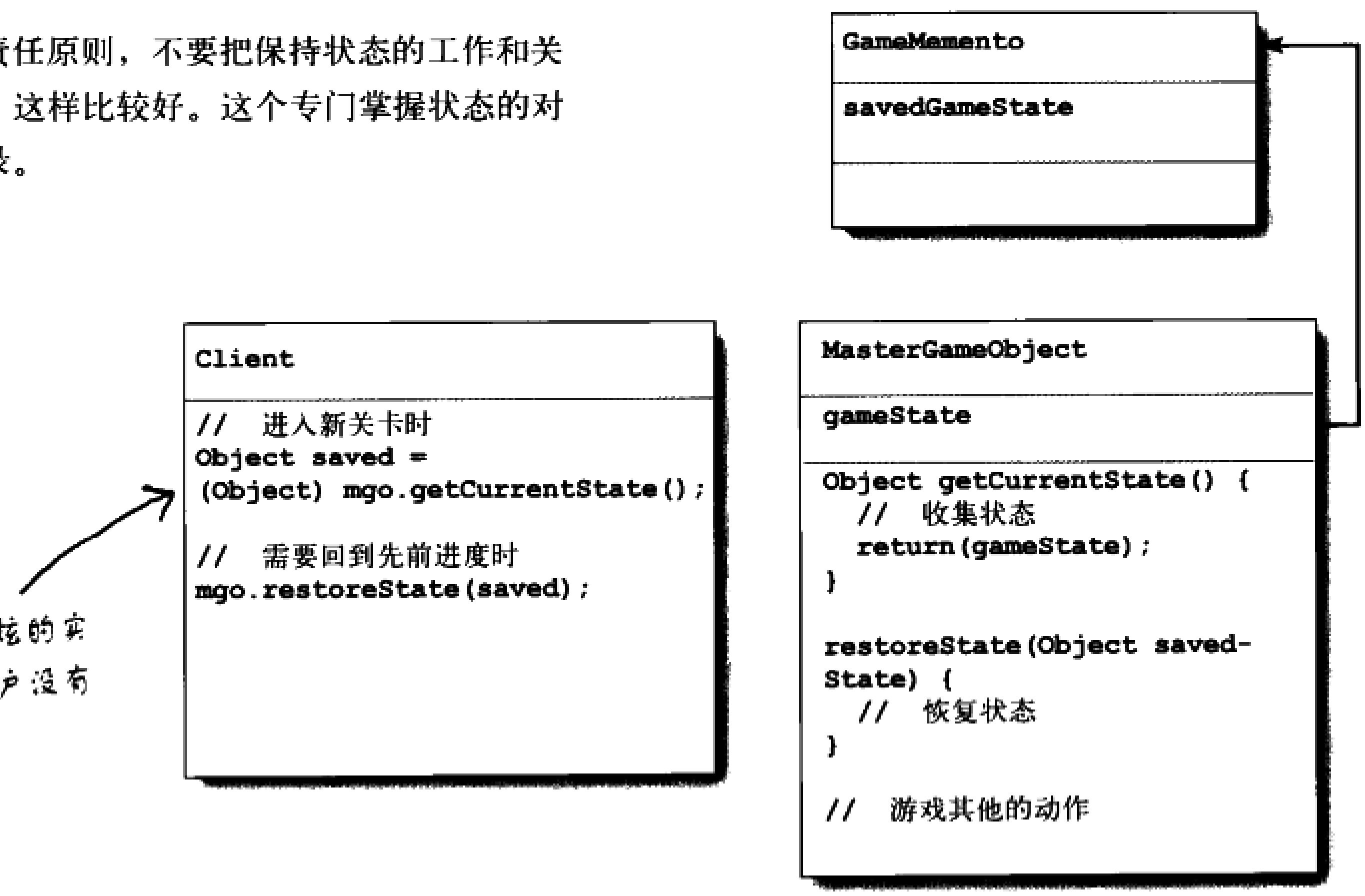


使用备忘录

备忘录模式有两个目标：

- 储存系统关键对象的重要状态。
- 维护关键对象的封装。

请不要忘记单一责任原则，不要把保持状态的工作和关键对象混在一起，这样比较好。这个专门掌握状态的对象，就称为备忘录。



尽管这不是一个很炫的实现，但是请注意客户没有接触到备忘录数据。

备忘录的优点

- 将被储存的状态放在外面，不要和关键对象混在一起，这可以帮助维护内聚。
- 保持关键对象的数据封装。
- 提供了容易实现的恢复能力。

备忘录的用途和缺点

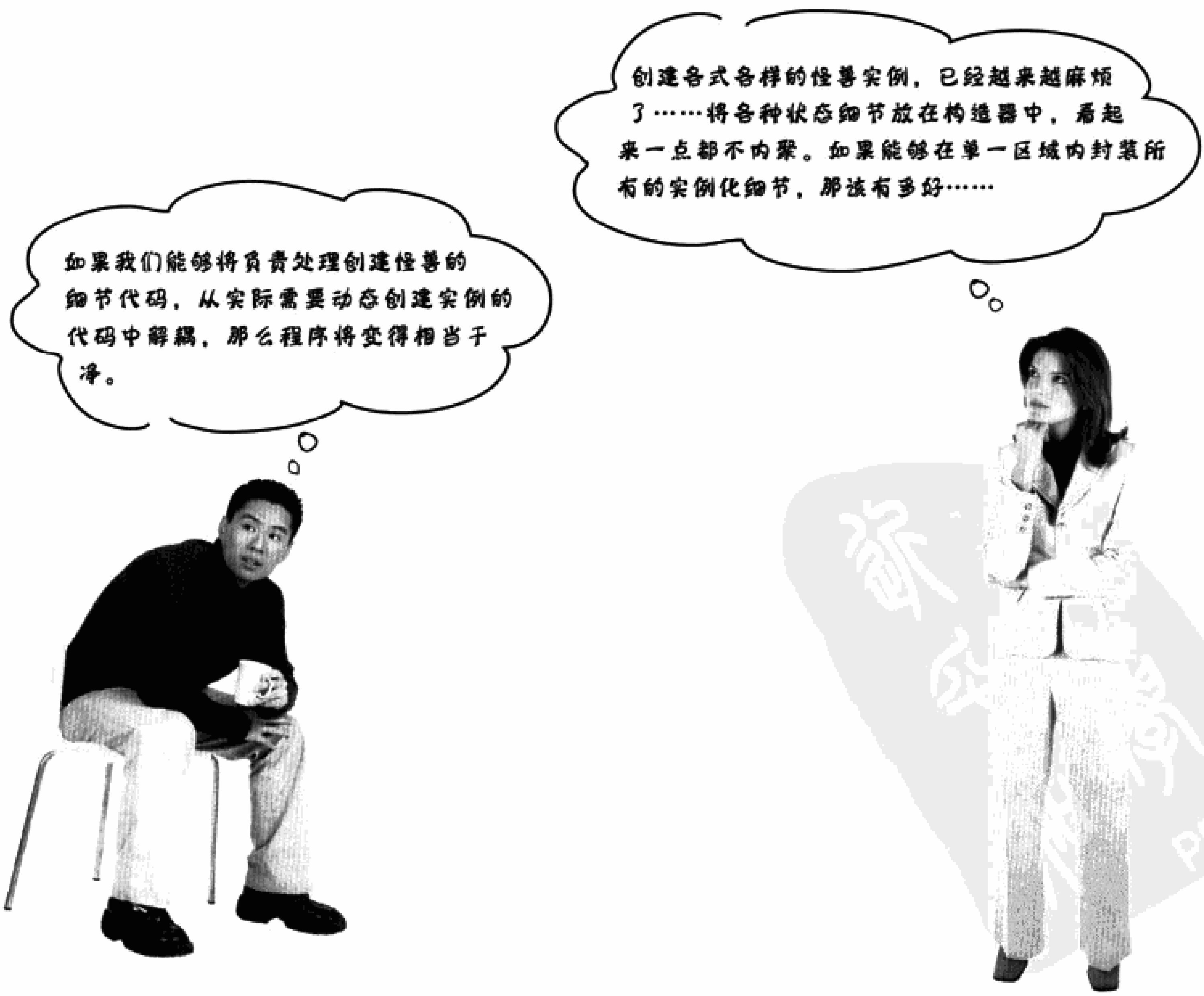
- 备忘录用于储存状态。
- 使用备忘录的缺点：储存和恢复状态的过程可能相当耗时。
- 在Java系统中，其实可以考虑使用序列化（serialization）机制储存系统的状态。

原型

当创建给定类的实例的过程很昂贵或很复杂时，就使用原型模式（Prototype Pattern）。

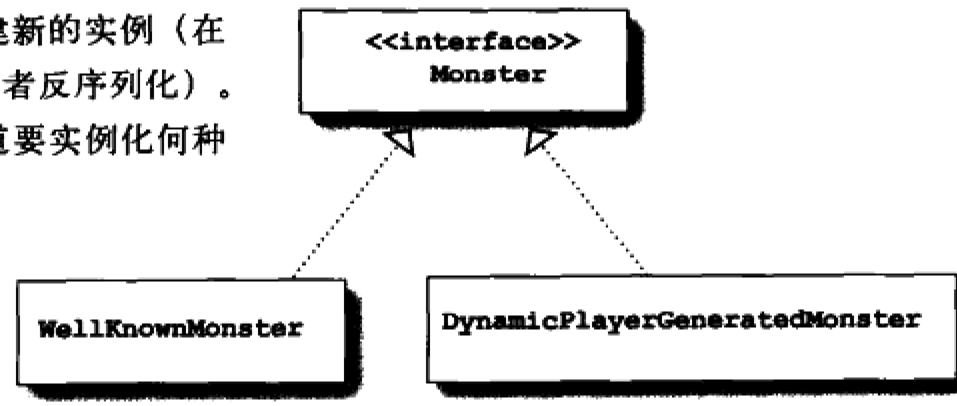
场景

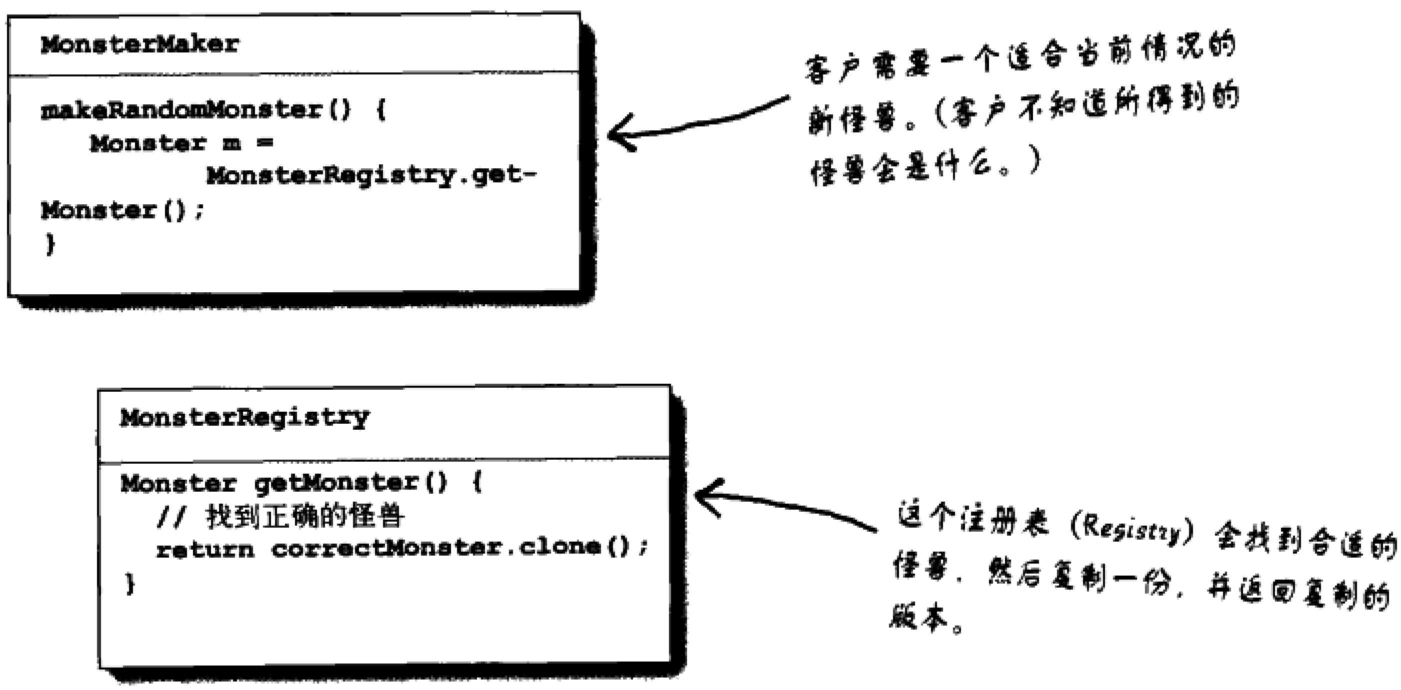
你的交互式角色扮演游戏中，怪兽有着贪得无厌的胃口。当英雄人物在动态创建的场景中闯荡时，遇到了庞大的敌军有待歼灭。你希望怪兽的特征能够随着场景的变换而演化。毕竟，如果让鸟一般的怪兽跟随你的角色进入海底世界，实在是没有道理。最后，你还希望能够让高级用户创建他们自己的怪兽。



原型来拯救你了

原型模式允许你通过复制现有的实例来创建新的实例（在Java中，这通常意味着使用clone()方法，或者反序列化）。这个模式的重点在于，客户的代码在不知道要实例化何种特定类的情况下，可以制造出新的实例。





原型的优点

- 向客户隐藏制造新实例的复杂性。
- 提供让客户能够产生未知类型对象的选项。
- 在某些环境下，复制对象比创建新对象更有效。

原型的用途和缺点

- 在一个复杂的类层次中，当系统必须从其中的许多类型创建新对象时，可以考虑原型。
- 使用原型模式的缺点：对象的复制有时相当复杂。

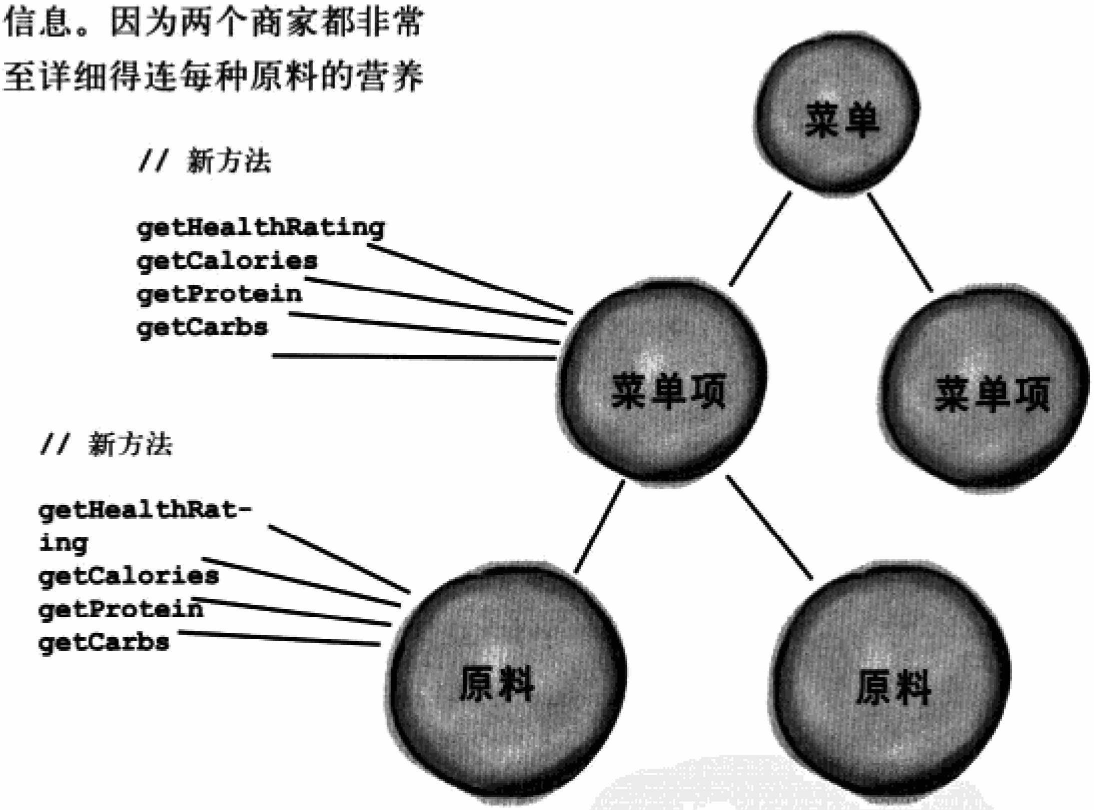
访问者

当你想要为一个对象的组合增加新的能力，且封装并不重要时，就使用访问者模式（Visitor Pattern）。

场景

对象村餐厅和对象村煎饼屋的常客，近来变得非常重视养生之道。在订餐之前，他们会询问营养信息。因为两个商家都非常愿意迎合顾客的需求，有些顾客甚至详细得连每种原料的营养成分也不放过。

Lou提出的解决方案……

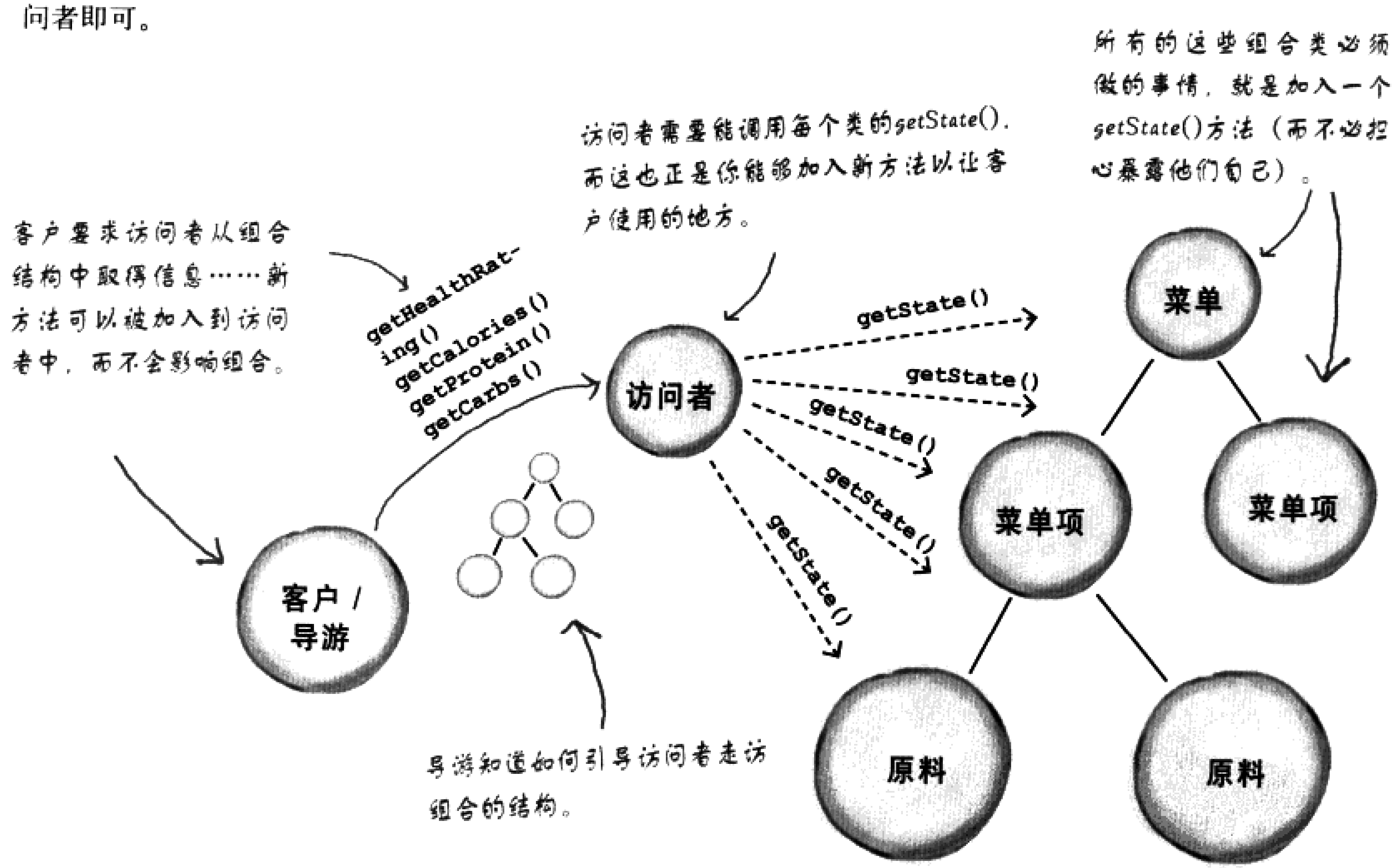


Mel的考虑……

"老天，看样子我们简直是打开了潘多拉的盒子。天晓得我们接下来要加入什么新方法，而每次一有新方法加入，就必须加到两个地方。还有，万一我们想要加强基本系统，比方说多了食谱类，那又该怎么办呢？我们就必须改变三个地方……"

访问者来访

访问者必须参观组合内的每个元素；这样的功能是在导游（Traverser）对象中，访问者通过导游的引导，收集组合中所有对象的状态。一旦状态被收集了，客户就可以让访问者对状态进行各种操作。当需要新的功能时，只要加强访问者即可。



访问者的优点

- 允许你对组合结构加入新的操作，而无需改变结构本身。
- 想要加入新的操作，相对容易。
- 访问者所进行的操作，其代码是集中在一起的。

访问者的用途和缺点

- 当采用访问者模式的时候，就会打破组合类的封装。
- 因为游走的功能牵涉其中，所以对组合结构的改变就更加困难。

索引

A

Abstract Factory Pattern（抽象工厂模式） 156. 参见 Factory Pattern（工厂模式）
Adapter Pattern（适配器模式）
  优点 242
  类适配器 244
  类图 243
  结合模式 504
  定义 243
  鸭子帖 245
  枚举 迭代器 适配器 248
  习题 251
  解释 241
  围炉夜话 247, 252～253
  介绍 237
  对象适配器 244
Alexander, Christopher 602
annihilating evil（歼灭恶势力） 606
Anti-Patterns（反模式） 606～607
  黄金榔头 607
application patterns（应用模式） 604
architectural patterns（架构模式） 604

B

Bridge Pattern（桥接模式） 612～613
Builder Pattern（生成器模式） 614～615
bullet points（要点） 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608
business process patterns（业务流程模式） 605

C

CD Cover Viewer（CD 封面浏览器） 463
Chain of Responsibility Pattern（责任链模式） 616～617
change（改变） 339
  提前 14
  软件开发的不变真理 8
  识别 53
Choc-O-Holic, Inc.（Choc-O-Holic公司） 175
class explosion（类爆炸） 81
code magnets（代码帖） 69, 179, 245, 350
cohesion（内聚） 339～340
Combining Patterns（结合模式） 500
  抽象工厂模式 508
  适配器模式 504
  类图 524
  组合模式 513
  装饰者模式 506
  观察者模式 516
Command Pattern（命令模式）
  类图 207
  命令对象 203
  定义 206～207
  介绍 196
  加载调用者 201
  日志请求 229
  宏命令 224
  空对象 214
  队列请求 228
  撤销 216, 220, 227

Composite Pattern（组合成模式）
与迭代器模式 368
类图 358
结合模式 513
组合行为 363
默认行为 360
定义 356
访谈 376～377
安全性 367
安全性vs.透明性 515
透明性 367, 375
composition（组合） 23, 85, 93, 247, 309
compound pattern（复合模式） 500, 522
controlling access（控制访问） 460. 参见
(Proxy Pattern代理模式)
creating objects（创建对象） 134
crossword puzzle（填字游戏） 33, 76, 163, 187, 231, 271, 310, 378, 490
cubicle conversation（办公室隔间对话） 55, 93, 195, 208, 387, 397, 433, 583～584

D

Decorator Pattern（装饰者模式）
与代理模式 472～473
类图 91
结合模式 506
办公室隔间对话 93
定义 91
缺点 101, 104
围炉夜话 252～253
访谈 104
介绍 88
在 Java I/O 中 100～101
结构型模式 591
Dependency Inversion Principle（依赖倒置原则） 139～143
好莱坞原则 298
Design Patterns（设计模式）
抽象工厂模式 156
适配器模式 243
优点 599
桥接模式 612～613
生成器模式 614～615
类目 589, 592～593
责任链模式 616～617
类模式 591
命令模式 206
组合模式 356
装饰者模式 91
定义 579, 581
发掘自己的模式 586～587
外观模式 264
工厂方法模式 134
绳量模式 618～619
解释器模式 620～621
迭代器模式 336
中介者模式 622～623
备忘录模式 624～625
空对象 214
对象模式 591
观察者模式 51
组织 589
原型模式 626～627
代理模式 460
简单工厂 114
单件模式 177
状态模式 410
策略模式 24

模板方法模式 289
使用 29
vs. 框架 29
vs. 库 29
访问者模式 628～629

Design Principles（设计原则） 参见（Object Oriented Design Principles面向对象设计原则）

Design Puzzle（设计迷题） 25, 133, 279, 395, 468, 542

Design Toolbox（设计工具箱） 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608

DJ View 534

domain specific patterns（领域特定模式） 604

E

Elvis（猫王） 526

encapsulate what varies（封装变化） 8～9, 75, 136, 397, 612

encapsulating algorithms（封装算法） 286, 289

encapsulating behavior（封装行为） 11

encapsulating iteration（封装迭代） 323

encapsulating method invocation（封装方法调用） 206

encapsulating object construction（封装对象构造） 614～615

encapsulating object creation（封装对象创建） 114, 136

encapsulating requests（封装请求） 206

encapsulating state（封装状态） 399

F

Facade Pattern（外观模式）
优点 260
与最少知识原则 269
类图 264
定义 264
介绍 258

Factory Method Pattern（工厂方法模式） 134. 参见 Factory Pattern

Factory Pattern（工厂模式）
Abstract Factory（抽象工厂）
与工厂方法 158～159, 160～161
类图 156～157
结合模式 508
定义 156
访谈 158～159
介绍 153
Factory Method（工厂方法）
优点 135
与抽象工厂 160～161
类图 134
定义 134
访谈 158～159
介绍 120, 131～132
再靠近一点 125
Simple Factory（简单工厂）
定义 117
介绍 114

family of algorithms（算法家族）. 参见
(Strategy Pattern 策略模式)

family of products（产品家族） 145

favor composition over inheritance（多用组合，少用继承） 23, 75

fireside chat（围炉夜话） 62, 247, 252, 308, 418, 472～473

Five minute drama（五分钟短剧） 48, 478

Flyweight Pattern（蝇量模式） 618-619

forces（力） 582

Friedman, Dan 171

G

Gamma, Erich 601
Gang of Four（四人组） 583, 601
Gamma, Erich 601
Helm, Richard 601
Johnson, Ralph 601
Vlissides, John 601
global access point（全局访问点） 177
gobble gobble（咯咯叫） 239
Golden Hammer（黄金榔头） 607
guide to better living with Design Patterns（以设计模式改善生活的指南） 578
Gumball Machine Monitor（糖果机监视器） 431

H

HAS-A（有一个） 23
Head First learning principles（学习原则） xxx
Helm, Richard 601
Hillside Group 603
Hollywood Principle, The（好莱坞原则） 296
与依赖倒置原则 298
Home Automation or Bust, Inc.（巴斯特家电自动化公司） 192
Home Sweet Home Theater（甜蜜家庭影院） 255
Hot or Not 475

I

inheritance（继承）
  缺点 5
  为了复用 5～6
  为了再利用 93
interface（接口） 12
Interpreter Pattern（解释器模式） 620～621
inversion（倒置） 141～142
IS-A（是一个） 23
Iterator Pattern（迭代器模式）
  优点 330
  与集合 347～349
  与组合模式 368
  与枚举 338
  与散列表 343, 348
  类图 337
  代码帖 350
  定义 336
  习题 327
  外部迭代器 338
  for/in 349
  内部迭代器 338
  介绍 325
  java.util.Iterator 332
  空迭代器 372
  多态迭代 338
  删除对象 332

J

Johnson, Ralph 601

K

KISS（Keep it simple 保持简单） 594

L

Law of Demeter. 参见（Principle of Least Knowledge 最少知识原则）
lazy instantiation（延迟实例化） 177
loose coupling（松耦合） 53

M

magic bullet（万灵丹） 594

master and student（大师与门徒） 23, 30, 85, 136, 592, 596
Matchmaking in Objectville（在对象村配对） 475
Mediator Pattern（中介者模式） 622～623
Memento Pattern（备忘录模式） 624～625
middleman（中间人） 237
Mighty Gumball, Inc.（万能糖果公司） 386
Model-View-Controller（模型－视图－控制器）
  与适配器模式 546
  与设计模式 532
  与 Web 549
  组合模式 532, 559
  介绍 529
  中介者模式 559
  观察者模式 532
  待烘烤代码 564～576
  歌 526
  策略模式 532, 545
  再靠近一点 530
Model 2 549 参见 （Model-View-Controller模型－视图－控制器）
  与设计模式 557～558
MVC 参见 （Model-View-Controller模型－视图－控制器）

N

Null Object（空对象） 214, 372

O

Objectville Diner（对象村餐厅） 26, 197, 316, 628
Objectville Pancake House（对象村煎饼屋） 316, 628
Object Oriented Design Principles（面向对象设计原则） 9, 30～31
  依赖倒置原则 139～143
  封装变化 9, 111
  多用合成，少用继承 23, 243, 397
  好莱坞原则 296
  一个类，一个责任 185, 336, 339, 367
  开放—关闭原则 86～87, 407
  最少知识原则 265
  针对接口编程，不针对实现编程 11, 243, 335
  让交互的对象之间尽量解耦 53
Observable（可观察者） 64, 71
Observer Pattern（观察者模式）
  类图 52
  代码帖 69
  结合模式 516
  办公室隔间对话 55
  定义 51～52
  围炉夜话 62
  五分钟短剧 48
  介绍 44
  在Swing中的用法 72～73
  Java 支持 64
  拉 63
  推 63
one-to-many relationship（一对多的关系） 51～52
OOPSLA 603
Open-Closed Principle（开放－关闭原则） 86～87
oreo 饼干 526
organizational patterns（组织模式） 605

P

part-whole hierarchy（部分—整体层次） 356. 参见 Composite Pattern
patterns catalog（模式类目） 581, 583, 585
Patterns Exposed（模式告白） 104, 158, 174, 377～378
patterns in the wild（荒野中的模式） 299, 488～489
patterns zoo（模式动物园） 604
Pattern Honorable Mention（模式荣誉奖） 117, 214
Pizza shop（比萨店） 112

Portland Patterns Repository 603
Principle of Least Knowledge（最少知识原则） 265～268
缺点 267
program to an implementation（针对实现编程） 12, 17, 71
program to an interface（针对接口编程） 12
program to an interface, not an implementation（针对接口编程，不针对现实编程） 11, 75
Prototype Pattern（原型模式） 626～627
Proxy Pattern（代理模式）
与适配器模式 471
与装饰者模式 471, 472～473
缓存代理 471
类图 461
定义 460
动态代理 474, 479, 486
与 RMI 486
习题 482
围炉夜话 472～473
java.lang.reflect.Proxy 474
保护代理 474, 477
代理动物园 488～489
待烘烤代码 494
远程代理 434
变体 471
虚拟代理 462
影像代理 464
publisher/subscriber（出版者/订阅者） 45

R

refactoring（重构） 354, 595
remote control（遥控器） 193, 209
Remote Method Invocation（远程方法调用）参见 RMI
remote proxy（远程代理） 434 (参见 Proxy Pattern代理模式)
reuse（复用） 13, 23, 85
RMI 436

S

shared vocabulary（共享词汇） 26～28, 599～600
sharpen your pencil（削尖你的铅笔） 5, 42, 54, 61, 94, 97, 99, 124, 137, 148, 176, 183, 205, 225, 242, 268, 284, 322, 342, 396, 400, 406, 409, 421, 483, 511, 518, 520, 589
Simple Factory（简单工厂） 117
SimUDuck（模拟鸭子） 2, 500
Singleton Pattern（单件模式）
优点 170, 184
与垃圾收集 184
与全局变量 185
与多线程 180～182
类图 177
定义 177
缺点 184
双重检查加锁 182
访谈 174
再靠近一点 173
Single Responsibility Principle（单一责任原则） 339. 参见 OO Design Principles: one class, one responsibility（面向对象设计原则：一个类，一个责任）
skeleton 440
Starbuzz Coffee（星巴兹咖啡） 80, 276
state machines（状态机） 388～389
State Pattern（状态模式）
与策略模式 411, 418～419
类图 410

定义 410
缺点 412, 417
介绍 398
共享状态 412
static factory（静态工厂） 115
Strategy Pattern（策略模式） 24
与状态模式 411, 418～419
与模板方法模式 308～309
封装行为 22
算法家族 22
围炉夜话 308
stub 440

T

Template Method Pattern（模板方法模式）
优点 288
与 Applet 307
与 java.util.Arrays 300
与策略模式 305, 308～309
与 Swing 306
与好莱坞原则 297
类图 289
定义 289
围炉夜话 308～309
挂钩 292, 295
介绍 286
再靠近一点 290～291
The Little Lisper（苏格拉底式的诱导问答） 171
thinking in patterns（用模式来思考） 594～595
tightly coupled（紧耦合） 53

U

undo（撤销） 216, 227
user interface design patterns（用户界面设计模式） 605

V

varies（改变）参见 encapsulate what varies（封装变化）
Visitor Pattern（访问者模式） 628～629
Vlissides,John 601

W

Weather-O-Rama（Weather-O-Rama气象站） 38
when not to use patterns（何时不使用模式） 596～598
Who Does What?（连连看） 202, 254, 298, 379, 422, 487, 588
Why a duck?（为何用鸭子？） 500
wrapping objects（包装对象） 88, 242, 252, 260, 473, 508 参见 Adapter Pattern, Decorator Pattern, Facade Pattern, Proxy Pattern

Y

your mind on patterns（使用模式的心智） 597

出版轶事



所有本书英文版内文排版都是由四位作者（Eric Freeman、Elisabeth Freeman、Kathy Sierra 和 Bert Bates）设计的。Kathy和 Bert原创 Head First系列的外观感觉。整本书由Adobe的 InDesign CS（简直不敢相信，这么酷的设计工具，我们还嫌不够）和Adobe的PhotoshopCS排版产生。本书的字体集使用了Uncle Skippy、Mister Frisky(你可能认为我们在开玩笑）、Ann Satellite、Baskerville、Comic Sans、Myriad Pro、Skippy Sharp、Savoye LET、Jokerman LET、Courier New和Wooddrow字体。

内文设计和制作都只在Apple Macintosh上进行。在Head First,我们都是“想得不同”（Think Different)（译注：苹果公司的宗旨）。所有Java代码都是使用James Gosling最喜欢的IDE来创建的，尽管我们真地应该试一试Erich Gamma的Eclipse。

在长时间的写作过程中，众多事物给我们提供能量：Honest Tea和Tejava(译注：都是饮料品牌）的咖啡因，圣达菲清新的空气，Babco De Gaia、Cocteau Twins、Buddha Bar I～VI、Delerium、Enigma、Mike Oldfield、Olive、Orb、Orbital、LTJ Bukem、Massive Attack、Steve Roach、Sasha and Digweed、Thievvery Corporation、Zero 7和Neil Finn的音乐（译注：以上提到的都是一些乐队和歌手的名字。Banco De Gaia是著名的电声乐队，Cocteau Twins的许多歌被王菲翻唱……）。

Head First
实验室

现在，最后来自Head First实验室的话……

我们世界级的优秀研究团队，正在疯狂的竞争中，发掘生命、宇宙和万物之谜……我们夜以继日地不停工作着，以免落于人后。

从未有过怀着如此高尚和远大目标的研究团队。目前，我致力于集合能量和脑力，以创造出“终极学习机”。一旦完美地创立后，你和其他人将可以加入我们的征途！

你很幸运能够拥有我们的第一个原型版本之一。但是，只有通过不断地精炼才能达到我们的目标。我们诚挚地邀请你——技术的先锋用户：定期报告你的进度到fieldreports@wickedlysmart.com。



万能糖果公司





没有你的帮助，我们的下一代，可能永远无法得知糖果机的乐趣。今天，一些没有弹性、贫乏的设计代码，危险地充斥在我们用Java编程的机器里。万能糖果公司不会让它发生。我们奉献自己协助你改进Java和OO设计的技巧，好让你帮我们建立次世代的万能糖果机。

来吧，Jave烤面包机已经是那么……久远(90年代)的事了，请到 http://www.mightygumball.com看看我们。（编注：这个网址不存在，千万别写E-mail 来质询。）

万能糖果公司

有糖果机的地方，
永远充满活力